Reference

Macro Assembler Version 2.00

Programming Family

IBM

Personal
Computer
Software

6361620

Macro Assembler
Version 2.00

Programming Family

Personal
Computer
Software

Second Edition (August 1984)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: International Business Machines Corporation provides this manual "as is," without warranty of any kind, either expressed or implied, including, but not limited to, the particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

This product could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer dealer.

The following paragraph applies only to the United States and Puerto Rico: A Reader's Comment Form is provided at the back of this publication. If the form has been removed, address comments to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

© Copyright International Business Machines Corporation 1981, 1984

About This Book

This manual contains detailed descriptions of the pseudo-ops and instructions used by the IBM Personal Computer Macro Assembler and Small Assembler. (See Chapter 1 for details on the format of each instruction.)

This Book is Organized as Follows

Chapter 1 "Introduction" discusses the format of the pseudo-ops and instructions. Information is also provided on the identity of the 8087, 8088, and 80286 instructions.

Chapter 2 "Pseudo Operations" describes the four types of pseudo-op instructions and the purpose and format of each pseudo-op. Examples are included.

Chapter 3 "Instruction Mnemonics" describes the purpose, format, logic, and encoding of the three sets of microprocessor instructions processed by the assembler. Examples are included.

Conventions Used In This Book

Certain conventions are used in this manual to define commands, statements, functions, and variables:

1. Words in CAPITAL LETTERS are keywords and must be entered as shown. You can enter either uppercase or lowercase characters.

2. You supply any items in lowercase *italic letters.*

3. Items in square brackets ([ ]) are optional.

   Note: Do not confuse this convention with the square bracket notation used in the "Data Transferring Operands" section of the "Assembly Language Format" chapter.

4. An ellipsis (...) indicates an item can be repeated as many times as you wish.

5. All punctuation except square brackets, such as commas, parentheses, angle brackets, slashes, or semicolons must be included where shown.

6. "Macro Assembler" is the large assembler that supports all of the instructions and pseudo-ops described in this book.

7. "Small Assembler" is the small assembler that requires less memory but does not support all of the instructions and pseudo-ops described in this book.

8. The term, "assembler," is used to refer to both the Macro Assembler and the Small Assembler.

Related Books

IBM Personal Computer *Macro Assembler* version 2.0, included with this product.

IBM Personal Computer Disk Operating System

IBM Personal Computer Technical Reference
We recommend version 2.02 or later for 8087 users.

Reference Material

Rector, Russell and George Alexy, *The 8086 Book, includes the 8088,* Osborne/McGraw-Hill, Berkeley, California, 1980.

The iAPX 86,88 User's Manual 210201, *The 8086 Family User's Manual* 9800722, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051.

For 8087 users; *The 8086 Family User's Manual Numerics Supplement* 121586, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051.

For 80286 users; *iAPX 286 Programmer's Reference Manual including the iAPX 286 Numeric Supplement* 210498, Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051.

Intel is a registered trademark of Intel Corporation
Microsoft is a trademark of Microsoft Corporation

Contents

Chapter 1. Introduction 1-1
Pseudo Operations 1-1
Conditional Pseudo-Ops 1-3
Data Pseudo-Ops 1-4
Listing Pseudo-Ops 1-4
Macro and Repeat Block Pseudo-Ops . 1-8
Mode Pseudo-Ops 1-17
Instructions 1-19
8087 Instructions 1-19
80286 Instructions 1-20
80287 Instructions 1-20
Instruction Fields 1-21
Flag Register 1-23
Instruction Symbols and Notation ... 1-23

Chapter 2. Pseudo Operations 2-1
.286C (Set 80286 Mode) 2-2
.8086 (Reset 80286 Mode) 2-3
.8087 (Set 8087-80287 Mode) 2-4
& Special Macro Operator 2-5
;; Special Macro Operator 2-6
! Special Macro Operator 2-7
% Special Macro Operator 2-8
ASSUME 2-9
COMMENT 2-11
.CREF .XCREF 2-12
DB Define Byte 2-13
DD Define Doubleword 2-15
DQ Define Quadword 2-17
DT Define Tenbytes 2-19
DW Define Word 2-21
ELSE 2-23
END 2-24
ENDIF 2-26
ENDM 2-27
ENDP 2-28
ENDS 2-29

EQU 2-30
= Equal Sign 2-31
EVEN 2-32
EXITM 2-33
EXTRN 2-34
GROUP 2-36
IFxxxx Conditional Pseudo-ops 2-40
INCLUDE 2-45
IRP 2-47
IRPC 2-49
LABEL 2-50
.LALL .SALL .XALL 2-52
.LFCOND (List False Conditionals) 2-53
.LIST .XLIST 2-54
LOCAL 2-55
MACRO 2-56
NAME 2-60
ORG 2-61
%OUT 2-62
PAGE 2-63
PROC 2-65
PUBLIC 2-67
PURGE 2-68
.RADIX 2-69
RECORD 2-71
REPT 2-75
SEGMENT 2-76
.SFCOND 2-80
STRUC 2-81
SUBTTL 2-83
.TFCOND 2-84
TITLE 2-85

Chapter 3. Instruction Mnemonics **3-1**
AAA ASCII Adjust for Addition 3-2
AAD ASCII Adjust for Division 3-3
AAM ASCII Adjust for Multiply 3-4
AAS ASCII Adjust for Subtraction 3-5
ADC Add with Carry 3-6
ADD Addition 3-8
AND Logical AND 3-11
BOUND (80286) Check Array Index Against Bounds 3-14

CALL Call a Procedure 3-16
CBW Convert Byte to Word 3-20
CLC Clear Carry Flag 3-21
CLD Clear Direction Flag 3-22
CLI Clear Interrupt Flag (Disable) 3-23
CMC Complement Carry Flag 3-24
CMP Compare Two Operands 3-25
CMPS/CMPSB/CMPSW Compare Byte or Word String 3-28
CWD Convert Word to Doubleword 3-31
DAA Decimal Adjust for Addition 3-32
DAS Decimal Adjust for Subtraction 3-33
DEC Decrement Destination by One 3-34
DIV Division, Unsigned 3-36
ENTER (80286) Make Stack Frame for Procedure Parameters 3-39
ESC Escape 3-41
F2XM1 (8087) 2 to the X power -1 3-42
FABS (8087) Absolute value 3-43
FADD (8087) Add Real 3-44
FADDP (8087) Add Real and Pop 3-47
FBLD (8087) Packed Decimal (BCD) Load 3-48
FBSTP (8087) Packed Decimal (BCD) Store and Pop 3-49
FCHS (8087) Change Sign 3-50
FCLEX (8087) Clear Exceptions 3-51
FCOM (8087) Compare Real 3-52
FCOMP (8087) Compare Real and Pop . . 3-55
FCOMPP (8087) Compare Real and Pop Twice 3-58
FDECSTP (8087) Decrement 8087 Stack Pointer 3-59
FDISI (8087) Disable Interrupts 3-60
FDIV (8087) Divide Real 3-61
FDIVP (8087) Divide Real and Pop 3-64
FDIVR (8087) Divide Real Reversed 3-65
FDIVRP (8087) Divide Real Reversed and Pop 3-68
FENI (8087) Enable Interrupts 3-69
FFREE (8087) Free Register 3-70
FIADD (8087) Integer Add 3-71
FICOM (8087) Integer Compare 3-72
FICOMP (8087) Integer Compare and Pop 3-74

FIDIV (8087) Integer Divide 3-76
FIDIVR (8087) Integer Divide Reversed . . 3-77
FILD (8087) Integer Load 3-78
FIMUL (8087) Integer Multiply 3-80
FINCSTP (8087) Increment 8087 Stack Pointer 3-81
FINIT (8087) Initialize Processor 3-82
FIST (8087) Integer Store 3-83
FISTP (8087) Integer Store and Pop 3-84
FISUB (8087) Integer Subtract 3-86
FISUBR (8087) Integer Subtract Reversed 3-87
FLD (8087) Load Real 3-88
FLD1 (8087) Load +1.0 3-90
FLDCW (8087) Load Control Word 3-91
FLDENV (8087) Load Environment 3-92
FLDL2E (8087) Load Log_2e 3-93
FLDL2T (8087) Load $\text{Log}_2 10$ 3-94
FLDLG2 (8087) Load $\text{Log}_{10} 2$ 3-95
FLDLN2 (8087) Load Log base e of 2 . . . 3-96
FLDPI (8087) Load PI 3-97
FLDZ (8087) Load Zero 3-98
FMUL (8087) Multiply Real 3-99
FMULP (8087) Multiply Real and Pop . . 3-102
FNCLEX (8087) Clear Exceptions 3-103
FNDISI (8087) Disable Interrupts 3-104
FNENI (8087) Enable Interrupts 3-105
FNINIT (8087) Initialize Processor 3-106
FNOP (8087) No Operation 3-107
FNSAVE (8087) Save State 3-108
FNSTCW (8087) Store Control Word . . . 3-109
FNSTENV (8087) Store Environment . . . 3-110
FNSTSW (8087) Store Status Word 3-112
FPATAN (8087) Partial Arc Tangent . . . 3-114
FPREM (8087) Partial Remainder 3-115
FPTAN (8087) Partial Tangent 3-117
FRNDINT (8087) Round to Integer 3-118
FRSTOR (8087) Restore State 3-119
FSAVE (8087) Save State 3-120
FSCALE (8087) Scale 3-121
FSQRT (8087) Square Root 3-122
FST (8087) Store Real 3-123
FSTCW (8087) Store Control Word 3-125
FSTENV (8087) Store Environment 3-126

FSTP (8087) Store Real and POP 3-127
FSTSW (8087) Store Status Word 3-129
FSUB (8087) Subtract Real 3-130
FSUBP (8087) Subtract Real and POP .. 3-133
FSUBR (8087) Subtract Real Reversed .. 3-134
FSUBRP (8087) Subtract Real Reversed and POP 3-138
FTST (8087) Test 3-139
FWAIT (8087) Wait (CPU Instruction) . 3-140
FXAM (8087) Examine 3-142
FXCH (8087) Exchange Registers 3-144
FXTRACT (8087) Extract Exponent and Significand 3-146
FYL2X (8087) Y * Log$_2$X 3-147
FYL2XP1 (8087) Y * Log$_2$(X+1) 3-148
HLT Halt 3-149
IDIV Integer Division, Signed 3-150
IMUL Integer Multiply 3-153
IMUL (80286) Integer Immediate Multiply 3-155
IN Input Byte or Word 3-157
INC Increment Destination by One 3-159
INS/INSB/INSW (80286) Input from Port to String 3-161
INT Interrupt 3-163
INTO Interrupt If Overflow 3-165
IRET Interrupt Return 3-167
J(condition) Jump Short If Condition Met 3-168
JMP Jump 3-171
LAHF Load AH from Flags 3-174
LDS Load Data Segment Register 3-175
LEA Load Effective Address 3-176
LEAVE (80286) High Level Procedure Exit 3-177
LES Load Extra Segment Register 3-178
LOCK Lock Bus 3-179
LODS/LODSB/LODSW Load Byte or Word String 3-180
LOOP Loop Until Count Complete 3-183
LOOPE/LOOPZ Loop If Equal/If Zero . 3-185
LOOPNE/LOOPNZ Loop If Not Equal/If Not Zero 3-187
MOV Move 3-189
MOVS/MOVSB/MOVSW Move Byte or

Word String 3-194
MUL Multiply, Unsigned 3-197
NEG Negate, Form Twos Complement .. 3-199
NOP No Operation 3-201
NOT Logical Not 3-202
OR Logical Inclusive Or 3-204
OUT Output Byte or Word 3-207
OUTS/OUTSB/OUTSW (80286) Output String to Port 3-209
POP Pop Word Off Stack to Destination . 3-211
POPA (80286) Pop All General Registers 3-213
POPF Pop Flags Off Stack 3-214
PUSH Push Word onto Stack 3-215
PUSH (80286) Push Immediate onto Stack 3-217
PUSHA (80286) Push All General Registers 3-218
PUSHF Push Flags onto Stack 3-219
RCL Rotate Left Through Carry 3-220
RCL (80286) Rotate Left Through Carry 3-223
RCR Rotate Right Through Carry 3-226
RCR (80286) Rotate Right Through Carry 3-229
REP/REPZ/REPE/ REPNE/REPNZ Repeat String Operation 3-232
RET Return from Procedure 3-235
ROL Rotate Left 3-238
ROL (80286) Rotate Left 3-240
ROR Rotate Right 3-243
ROR (80286) Rotate Right 3-245
SAHF Store AH in Flags 3-248
SAL/SHL Shift Arithmetic Left/Logical Left 3-249
SAL/SHL (80286) Shift Arithmetic Left/Shift Logical Left 3-252
SAR Shift Arithmetic Right 3-255
SAR (80286) Shift Arithmetic Right 3-257
SBB Subtract with Borrow 3-260
SCAS/SCASB/SCASW Scan Byte or Word String 3-263
SHR Shift Logical Right 3-266
SHR (80286) Shift Logical Right 3-268
STC Set Carry Flag 3-271
STD Set Direction Flag 3-272
STI Set Interrupt Flag (Enable) 3-273

STOS/STOSB/STOSW Store Byte or Word String 3-274
SUB Subtract 3-277
TEST Test (Logical Compare) 3-280
WAIT Wait 3-283
XCHG Exchange 3-284
XLAT Translate 3-286
XOR Exclusive OR 3-287

Index **Index-1**

Chapter 1. Introduction

Pseudo Operations

Pseudo operations, also called pseudo-ops, tell the assembler what to do with conditional branches, data, listings, and macros. Pseudo operations are sometimes called directives to the assembler. Pseudo-ops (except for certain Define Data) *do not* produce machine language code. They have mnemonics similar to the mnemonics of the assembler instructions.

Each pseudo-op is described in detail in Chapter 2 "Pseudo-op Operations" in this book. The pseudo-ops are arranged in alphabetic order for ease of reference in that chapter.

For the purpose of describing general information, this chapter groups the pseudo-ops by type. The five types of pseudo-ops are:

- Conditional
- Data
- Listing
- Macro
- Mode

Type	Pseudo-ops		
Conditional	ELSE	IFDEF	IFNB
	ENDIF	IFDIF	IFNDEF
	IF	IFE	IF1
	IFB	IFIDN	IF2
Data	ASSUME	ENDS	NAME
	COMMENT	EQU	ORG
	DB	= (Equal Sign)	PROC
	DD		PUBLIC
	DQ	EVEN	.RADIX
	DT	EXTRN	RECORD
	DW	GROUP	SEGMENT
	END	INCLUDE	STRUC
	ENDP	LABEL	
Listing	.CREF	PAGE	TITLE
	.LALL	.SALL	.XALL
	.LFCOND	.SFCOND	.XCREF
	.LIST	SUBTTL	.XLIST
	%OUT	.TFCOND	
Macro	ENDM	IRPC	PURGE
	EXITM	LOCAL	REPT
	IRP	MACRO	
Mode	.286C	.8086	.8087

Conditional Pseudo-Ops

Conditional Pseudo-ops		
ELSE	IFDEF	IFNB
ENDIF	IFDIF	IFNDEF
IF	IFE	IF1
IFB	IFIDN	IF2

The conditional pseudo-ops can be nested to any level. They are not limited to usage within a macro. Any operand of a conditional must be known on pass 1 to avoid errors and incorrect evaluation. Each IFxx pseudo-op must be matched with a corresponding ENDIF pseudo-op. All conditional pseudo-ops use the format:

```
IFxx operand
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

Note: Do not confuse the assembly time conditional pseudo-ops IFxx, ELSE, and ENDIF with the execution time conditional execution Structure statements $IF, $ELSE, and $ENDIF used by SALUT. See Chapter 4 “SALUT” in the *IBM Personal Computer Macro Assembler* book for more detail on $IF, $ELSE, and $ENDIF.

Data Pseudo-Ops

Data Pseudo-ops

ASSUME
COMMENT
DB
DD
DQ
DT
DW
END
ENDP
ENDS
EQU
= (Equal Sign)
EVEN
EXTRN
GROUP
INCLUDE
LABEL
MASK
NAME
ORG
PROC
PUBLIC
.RADIX
RECORD
SEGMENT
STRUC
WIDTH

Refer to the individual data pseudo-op in Chapter 2 "Pseudo-op Operations".

Listing Pseudo-Ops

Listing Pseudo-ops

.CREF
.LALL
.LFCOND
.LIST
%OUT
PAGE
.SALL
.SFCOND
SUBTTL
.TFCOND
TITLE
.XALL
.XCREF
.XLIST

False Conditional Blocks

The listing of false conditional blocks is controlled by the pseudo-ops: .LFCOND (List False Conditionals), .SFCOND (Suppress False Conditionals), and .TFCOND (Toggle False Conditionals).

In the following example, one or the other of the %OUT statements is a false conditional block depending on which pass the assembler is executing.

```
IF2
 %OUT END OF IBM VERSION
ELSE
 %OUT START OF IBM VERSION
ENDIF
```

The assembler can either list or suppress the listing of false conditional blocks of an IF conditional pseudo-op. The decision to list or suppress is based on several conditions.

The default mode says that in the absence of explicit directions, the default mode controls the option of "suppress or print" false conditionals. At assembler start time, the default mode is to "suppress" the printing of false conditionals. However, an optional parameter on the assembler command line can be used to request the opposite default definition. The /X parameter sets the default to "list" false conditionals.

Once the assembly has started, the definition of the default condition can be toggled, that is , switched to its opposite condition, by including in the source file the pseudo-op, .TFCOND, "Toggle False Condition" If the default condition has been "suppress", then the .TFCOND changes the default to "list". On the other hand, if the default condition has been "list", the .TFCOND sets the default to "suppress".

Putting a .TFCOND at the start of the source produces the same effect as entering the /X parameter on the assembler command line. However, having .TFCOND at the start of the listing, AND the /X on the command line produces the same effect as doing neither the pseudo-op nor the parameter.

Instead of allowing the default state control the printing of false conditionals, the programmer can expressly state, for a particular section of the code, that the default state is to be ignored and false conditionals are to be "listed" by including within the source the pseudo-op, .LFCOND, "List False Conditionals". Similarly, false conditionals can be directed to be "suppressed" regardless of the default state by expressly using in the source the pseudo-op, .SFCOND, "Suppress False Conditionals".

At the end of the particular section of code whose false conditionals are either to be listed or suppressed as commanded by the .LFCOND or .SFCOND pseudo-ops, the assembler can be told to return to its default state regarding false conditionals by including within the source to pseudo-op, .TFCOND. This pseudo-op however, not only indicates the end of the section expressly controlled, but also, as previously stated, toggles the default state. If it was desired to merely terminate the section under express control and just return to the default state, it is necessary to issue the .TFCOND twice, the second one toggling the default state definition back like it was.

> **Note:** The listing of false conditionals defined within MACROS will always be suppressed unless the .LALL pseudo-op is used. For the /X parameter, .LFCOND, .SFCOND, and .TFCOND to be effective within MACROS the .LALL pseudo-op must also be used.

This chart shows the effects of the three pseudo-ops when encountered with the /X and no /X parameter given in the assembler command line:

PSEUDO-OP	/X	NO/X
(none)	LIST	SUPPRESS
⋮		
.SFCOND	SUPPRESS	SUPPRESS
⋮		
.LFCOND	LIST	LIST
⋮		
.TFCOND	SUPPRESS	LIST
⋮		
.TFCOND	LIST	SUPPRESS
⋮		
.SFCOND	SUPPRESS	SUPPRESS
⋮		
.TFCOND	SUPPRESS	LIST
⋮		
.TFCOND	LIST	SUPPRESS
⋮		
.TFCOND	SUPPRESS	LIST

Note: The .SFCOND and .LFCOND pseudo-ops are absolute overrides. The .TFCOND pseudo-op toggles the default state.

Macro and Repeat Block Pseudo-Ops

Macro Pseudo-ops		
ENDM	IRPC	PURGE
EXITM	LOCAL	REPT
IRP	MACRO	

The macro facilities provided by the Macro assembler include: MACRO, REPT, IRP, and IRPC. Each of these four macro operations is ended by the ENDM pseudo-op.

> Note: These functions are not supported by the Small Assembler (ASM).

Terms Used

These terms are used with the MACRO, REPT, IRP, and IRPC pseudo-ops:

- *dummy* is used to represent a dummy parameter. All dummy parameters are valid symbols that appear in the body of a macro expansion.

- *dummylist* is a list of *dummies* separated by commas.

- <*operandlist*> is a list of operands separated by commas. The <*operandlist*> must be delimited by angle brackets. Two angle brackets with no intervening character entries (<>) or two commas with no intervening character entries becomes a null operand in the list. An operand is a character or series of characters ended by a comma or the end angle bracket (>). With angle brackets that are nested inside an <*operandlist*>, one level of brackets is removed each time the bracketed operand is used in an <*operandlist*>. A quoted string is an acceptable operand and is passed as such.

- A *parmlist* is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is ended by the end of line or a comment).

A *macro* is a collection of assembler statements that can appear several times in a program with some optional modifications each time it is used.

Using A Macro

To properly use a macro, you must:

- Define a macro using the MACRO and ENDM pseudo-ops.
- Invoke the macro by using the name of the macro just as if it were an op-code.

The definition must be done before the invocation. Once defined, a macro can be used, then redefined.

Defining A Macro

To define a macro, you must first understand what are the various parts of a MACRO definition:

- The *beginning* of a macro definition is the MACRO pseudo-op.
- The *end* of a macro definition is the ENDM pseudo-op.
- The *body* of a macro is all the lines of assembler statements between MACRO and ENDM.
- The *name* of the macro (the pseudo-op that is used to invoke it) is defined in the name field of the MACRO pseudo-op.

- The *parameters* (optional) are a series of one or more dummy symbol names listed as operands on the MACRO pseudo-op. If more than one parameter is used, the dummy symbols must be separated by commas or blanks. In some cases, there are no parameters. The number of parameters is limited only by how many you can define on a 132 character MACRO pseudo-op statement.

Macro Parts

The body of the macro can have several parts:

- LOCAL—If this pseudo-op is used, it must be the very next statement following the MACRO pseudo-op. No comment, nor even a blank line, may appear between MACRO and LOCAL. You need LOCAL only if there are labels defined in the body of the macro, which would produce a "Duplicate labels" error if the macro were invoked (expanded) more than once. LOCAL substitutes a unique label for the one you have used, so each time the macro is invoked a different label is generated and therefore avoids the duplication problem. Multiple consecutive LOCAL statements are allowed.

- Remarks or Comments—Indicated either by the COMMENT pseudo-op or by the semicolon. They are listed in a macro expansion only if under the .LALL mode. A comment begun with the double semicolon is never listed when the macro is invoked.

- Pseudo-ops and Instructions—These statements become effective only when the MACRO containing them is invoked. For example, a MACRO definition can be contained within the body of another macro. The inner macro is not defined until the outer macro is invoked as shown in the following example.

```
TITLE SAMPLE MACRO REDEFINING ITSELF
.LALL   ;set print mode
;        so (;) comments print
CALLSUB MACRO
        JMP     SHORT SKIP
;;Generate the subroutine,
;; one time only
SUBRT   PROC    NEAR

;   (body of subroutine here)
        RET     ;return from subrt
;                to main caller
SUBRT   ENDP
SKIP:   CALL    SUBRT
;Redefine the 'callsub' macro so
; SUBRT is generated once only.
   CALLSUB MACRO
   CALL    SUBRT
           ENDM ;end of inner macro

        ENDM    ;end of outer macro
;**********************************
CSEG    SEGMENT
        ASSUME  CS:CSEG
MAIN      PROC  FAR
          CALLSUB  ;generate subrt,
;                    then call it
          CALLSUB  ;just call subrt
MAIN      ENDP
CSEG    ENDS
        END
```

Implementing Macro Libraries

The INCLUDE facility provides an excellent way of implementing macro libraries. You should put the macros in a source file that contains only the macro definitions and place an INCLUDE statement referencing the macro library near the start of your source program.

It speeds-up operations and print time if the INCLUDE is executed only on pass 1 by using the conditional pseudo-op IF1 as:

```
IF1
  INCLUDE  maclib.mac
ENDIF
```

The macro definitions are not listed (since printing the listing is a pass 2 operation). Also, on pass 2, this file is not opened and read again, which is an unnecessary operation. This use of IF1 before the INCLUDE is not allowed if:

- The library contains source statements that actually generate code.
- MACRO redefined itself on the first invocation as shown in the preceding example.

However, the use of IF1 is useful if the library has macro definitions only. (The library can also have the definitions of structures and records, but no expansions of these.)

The parameters are defined on the MACRO pseudo-op separated by commas. These parameters are position-dependent.

```
FOO MACRO P1,P2,P3
```

P1 is known as the first parameter, P2 the second, and P3 the third.

These symbols, defined as parameters, can appear anywhere in the body of the macro.

Examples of instructions using the parameters within a macro are:

```
MOV  AX,P1
P2   P3
```

When this macro is invoked (expanded), the invocation statement pseudo-op is whatever has been defined in the *name* field of a previously defined MACRO pseudo-op. The invocation statement cannot have a name field, but can specify actual parameters that correspond by relative position to the dummy parameters defined in the macro definition. For example:

```
FOO   WORDVAR,INC,AX
```

In this example, WORDVAR is the first parameter, INC is the second, and AX is the third. The macro processor uses these specific character strings as direct replacements for the dummy parameters, assigning:

P1 is replaced by WORDVAR
P2 is replaced by INC
P3 is replaced by AX

Now the instructions within the body of the macro are generated, showing the above substitutions, and become:

```
MOV   AX,WORDVAR
INC   AX
```

One parameter was used as a variable name; another as an opcode; and the last as a register name.

If a dummy parameter within the body of a macro is not separated by proper delimiters, it is not recognized as a dummy parameter and is not substituted. If the body had this statement:

```
JMP TAP1
```

The symbol TAP1 is considered a label. The P1 portion is not replaced by the first parameter taken from the invocation statement. However, a dummy parameter can be recognized within a string by prefixing it with the ampersand (&) sign. For example:

```
JMP TA&P1
```

This time the specified operand is substituted, generating:

```
JMP TAWORDVAR
```

An instruction op-code can be built by joining together several pieces in this manner using the ampersand. For example:

```
LEAP MACRO   COND,LAB
     J&COND  LAB
     ENDM
```

When invoked by:

```
LEAP  Z,THERE
```

Generating:

```
JZ  THERE
```

The equal sign (=) pseudo-op is also helpful within a macro because it can be used to redefine the value of a counter. For example:

```
BUMP    MACRO
CNTR    =       CNTR+1
        ENDM
```

Note: CNTR should have been initialized before invoking BUMP.

This counter has a value that can be accessed by using the special prefix, %, before the parameter in a macro invocation statement:

```
ERRMSG    MACRO  TEXT,V,X
CNTR      =      CNTR+1
          MSG    %CNTR,TEXT
          ENDM
;
MSG       MACRO  COUNT,STRING
MSG&COUNT DB STRING
          ENDM
```

Note: One macro calling another.

Invoke the first macro by:

```
ERRMSG  'SYNTAX ERROR'
```

Which generates:

```
MSG1 DB 'SYNTAX ERROR'
```

Invoke it again with:

```
ERRMSG  'INVALID OPERAND'
```

Which generates:

```
MSG2 DB 'INVALID OPERAND'
```

The % prefix has caused a different kind of substitution: a substitution of the "value" of a parameter, not the character string of that parameter's name.

On closer examination of your results, you may wonder about what happened to the CNTR = CNTR+1 statement, since the macro expansion does not show anything for this line. Yet apparently, the CNTR variable is being incremented, as shown by the two variable names, MSG1 and MSG2. For a solution, try putting .LALL at the top of the source and all becomes apparent. The previous try was done in the .XALL mode (by default), which suppresses the listing of any statement that does not actually generate code.

Mode Pseudo-Ops

Mode Pseudo-ops		
.286C	.8086	.8087

The mode facilities provided by the Macro Assembler include the /R and /E parameters and the .286C, .8086, and .8087 pseudo-ops. The Mode pseudo-ops, and the /R and /E parameters, allow the IBM Personal Computer Macro Assembler to support various machine configurations and microprocessors.

Notes:

1. See Chapter 2 "Pseudo-op Operations" in this book for detail on each of the Mode pseudo-ops and Chapter 5 "Macro Assembler" in the *IBM Personal Computer Macro Assembler* book for detail on the parameters.
2. These functions are not supported by the Small Assembler (ASM).

The IBM Personal Computer Macro Assembler can assemble instructions for various machine configurations and micro processors.

For example:

- 8088-based IBM Personal Computers
- 8088-based IBM Personal Computers with the 8087 Math Co-Processor feature
- 80286-based IBM Personal Computers
- 80286-based IBM Personal Computers with the 80287 Math Co-Processor feature

The instructions are considered to be upward-compatible at source code level and, generally upward-compatible at object code level. This means that the "80286 only" type of instructions are not compatible with the 8088. However, the 8088 type instructions are compatible with the 80286. The IBM Personal Computer Macro Assembler can recognize the different types of instructions. The Mode Pseudo-ops tell the Macro Assembler what type of instructions to accept.

The default Mode is 8088/8086. In this mode, the 8087 and 80286 instructions cause a syntax error during assembly. A .286C mode pseudo-op is required for the Macro Assembler to recognize 80286 type instructions. A .8087 mode pseudo-op, /E, or /R parameter is required for the Macro Assembler to recognize 8087 type instructions.

Instructions

Each instruction is described in detail in Chapter 3 "Instruction Mnemonics" of this book. The instructions are arranged in alphabetic order for ease of reference in that chapter.

8087 Instructions

There are some instructions available for the 8088 based IBM Personal Computers with the 8087 Math Co-Processor feature. The 8087 instructions can be recognized by their headings in Chapter 3. Each instruction that can only be used on a 8087 IBM Personal Computer Math Co-Processor is shown by the number following its name at the top of each page, and also in the table of contents for this book as well as in the index. The instruction is followed by (8087) showing it as an 8087 instruction.

For example:

- **FADD (8087) / Add Real**
- **FDIV (8087) / Divide Real**
- **FIST (8087) / Integer Store**

Note: All 8087 instructions begin with the letter "F" making them easer to distinguish from other instructions.

80286 Instructions

The 80286 has two modes of operation, they are:

- Real Address Mode
- Protected Virtual Address Mode

The IBM Personal Computer Macro Assembler supports the Real Address Mode subset of the 80286 instruction set.

The 80286 instructions can be recognized by their headings in Chapter 3. Each instruction that can only be used on a 80286-based IBM Personal Computer is shown by the number following its name at the top of each page, and also in the table of contents for this book as well as in the index. The instruction is followed by (80286) showing it as an 80286 instruction.

For example:

- **PUSHA (80286) / Push All General Registers**
- **POPA (80286) / Pop All General Registers**
- **INS INSB INSW (80286) / Input from Port to String**

Some of the (80286) instructions are enhanced variations of 8088 instructions. For example, compare IMUL with IMUL (80286) in Chapter 3 of this book.

80287 Instructions

The 80287 uses the same instruction set as the 8087.

Instruction Fields

In Chapter 3, the "Encoding" field shows the assembled code for the instruction. There are 1 to 4 bytes represented, depending on the particular instruction or form of the instruction.

The general format of an instruction after assembled is:

operation-code-byte [*addressing-mode-byte*[*disp(s)*]]

The *operation-code* determines if there is an *address-mode-byte* and this in turn determines if *displacements* are added to the end of the instruction.

Operation Code Byte

See the individual instructions in Chapter 3.

Addressing Mode Byte

The second instruction byte, if required, is the addressing mode byte (*modregr/m*) consisting of the *mode field*, *register field*, and *register/memory field.*

Mode Field

If mod=00,	then disp=0, disp-low and disp-high are absent (unless, mod=00 and r/m=110), then EA=disp-high and disp-low.
If mod=01,	then disp=disp-low sign extended to 16 bits, disp-high is absent.
If mod=10,	then disp=disp-high and disp-low.
If mod=11,	then r/m is a reg field.

Register Field

Each of the general 8-bit, general 16-bit, base (BX and BP), and index (SI and DI) registers can be used in arithmetic and logical operations.

The 3-bit binary code assignment for the general, base and index registers and the 2-bit binary code assignment for the segment registers are:

8-Bit Registers (w = 0)	16-Bit Registers (w = 1)	Segment Registers
AL = 000	AX = 000	ES = 00
CL = 001	CX = 001	CS = 01
DL = 010	DX = 010	SS = 10
BL = 011	BX = 011	DS = 11
AH = 100	SP = 100	
CH = 101	BP = 101	
DH = 110	SI = 110	
BH = 111	DI = 111	

Register/Memory Field

If r/m = 000, then EA = [BX] + [SI] + disp
If r/m = 001, then EA = [BX] + [DI] + disp
If r/m = 010, then EA = [BP] + [SI] + disp
If r/m = 011, then EA = [BP] + [DI] + disp
If r/m = 100, then EA = [SI] + disp
If r/m = 101, then EA = [DI] + disp
If r/m = 110, then EA = [BP] + disp
If r/m = 110, and mod = 00,
then EA = disp-high and disp-low
If r/m = 111, then EA = [BX] + disp

Flag Register

```
X X X X OF DF IF TF SF ZF X AF X PF X CF
15                 8 7                 0
```

X = Reserved

CF = Carry Flag
PF = Parity Flag
AF = Auxiliary Carry Flag
ZF = Zero Flag
SF = Sign Flag

TF = Trap Flag
IF = Interrupt Flag
DF = Direction Flag
OF = Overflow Flag

Instruction Symbols and Notation

The symbols and notations used to describe the instructions are listed below:

Instruction symbols	Definition
AX	Word Accumulator (16-bit), that can be split and addressed as two 8-bit registers (AH and AL).
AH	High-order byte of Word Accumulator (AX).
AL	Byte Accumulator (low-order byte of Word Accumulator AX).
BX	Data Base Register (16-bit), that can be split and addressed as two 8-bit registers (BH and BL).
BH	High-order byte of BX register.

Instruction symbols	Definition
BL	Low-order byte of BX register.
CX	Count Register (16-bit), that can be split and addressed as two 8-bit registers (CH and CL).
CH	High-order byte of CX register.
CL	Low-order byte of CX register.
DX	Data Register (16-bit), that can be split and addressed as two 8-bit registers (DH and DL).
DH	High-order byte of DX register.
DL	Low-order byte of DX register.
SP	Stack Pointer (16-bit).
BP	Stack Base Pointer (16-bit).
IP	Instruction Pointer (16-bit).
Flags	16-bit register word, where nine flags reside.
DI	Destination index register (16-bit).
SI	Source Index register (16-bit).

Instruction symbols	Definition
CS	Code Segment register (16-bit).
DS	Data Segment register (16-bit).
ES	Extra Segment register (16-bit).
SS	Stack Segment register (16-bit).
REG8	The name or encoding of an 8-bit CPU register location.
REG16	The name or encoding of a 16-bit CPU register location.
LSRC,RSRC	Refer to operands of an instruction, generally left *source* and right *source* when two operands are used. The left-most operand is also called the *destination* operand, and the right-most is called the *source* operand.
modregr/m	The second byte of the instruction (usually identifies the instruction's operands).
mod	Bits 7, 6 of the modregr/m byte. This 2-bit field defines the addressing mode.

Instruction symbols	Definition
reg	Bits 5, 4, 3 of the modregr/m byte. This 3-bit field usually specifies REG8 or REG16 in the description of an instruction, or three op-code bits of an 8087 instruction.
r/m	Bits 2, 1, 0 of the modregr/m byte used in accessing memory operands. This 3-bit field defines EA with the mode and w fields.
ST	The 8087 current top of stack register.
ST(*i*)	An 8087 register stack element that is the *i*th register below the current top of stack register.
(*i*)	Bits 2, 1, 0 of the modregr/m byte used in accessing 8087 register stack element operands.
EA	Effective address (16-bit).
w	A 1-bit field in an instruction, identifying byte instructions (w=0), and word instructions (w=1).
s	If s=0, 2-byte immediate. If s=1, 1-byte immediate; sign bit extended to 16 bits.

Instruction symbols	Definition
d	A 1-bit field, "d" identifies direction. For d=0, register is the *source*. For d=1, register is the *destination*.
(...)	Parentheses mean the contents of the enclosed register or memory location.
(BX)	The contents of register BX, is the address where an 8-bit operand is located.
((BX))	Means this 8-bit operand, the contents of the memory location pointed at by the contents of register BX.
(BX) +1:(BX)	Means the address (of a 16-bit operand) whose low-order 8-bits reside in the memory location pointed at by the contents of register BX and whose high-order 8-bits reside in the next sequential memory location, (BX) + 1.
((BX) +1:(BX))	Means the 16-bit operand that resides there.
Concatenation, for example, ((DX) +1:(DX))	Means a word is the concatenation of two 8-bit bytes, the low-order byte in the memory location pointed at by DX and the high-order byte in the next sequential memory location.

Instruction symbols	Definition
addr	Address (16-bit) of a byte in memory.
addr-low	Least significant byte of an address.
addr-high	Most significant byte of an address.
addr +1:addr	Addresses of two consecutive bytes in memory, beginning at addr.
data	Immediate operand (8-bit if w=0; 16-bit if w=1).
data-low	Least significant byte of 16-bit data word.
data-high	Most significant byte of 16-bit data word.
disp	Displacement.
disp-low	Least significant byte of 16-bit displacement.
disp-high	Most significant byte of 16-bit displacement.
←	Assignment.
=	Assignment or equals.
+	Addition.
-	Subtraction.
*	Multiplication.

Instruction symbols	Definition
/	Division.
%	Modulo.
&	AND.
IOR	Inclusive OR.
XOR	Exclusive OR.

Chapter 2. Pseudo Operations

The pseudo-ops are arranged in this chapter in alphabetic order for ease of reference. The following table shows the different types of pseudo-ops.

Refer to Chapter 1 in this book for general information on Pseudo Operations.

Type	**Pseudo-ops**		

Conditional	ELSE
	ENDIF
	IF
	IFB
		IFDEF
		IFDIF
		IFE
		IFIDN
			IFNB
			IFNDEF
			IF1
			IF2

Data	ASSUME COMMENT DB DD DQ DT DW END ENDP	ENDS EQU = (Equal Sign) EVEN EXTRN GROUP INCLUDE LABEL	NAME ORG PROC PUBLIC .RADIX RECORD SEGMENT STRUC
Listing	.CREF .LALL .LFCOND .LIST %OUT	PAGE .SALL .SFCOND SUBTTL .TFCOND	TITLE .XALL .XCREF .XLIST
Macro	ENDM EXITM IRP	IRPC LOCAL MACRO	PURGE REPT
Mode	.286C	.8086	.8087

.286C
(Set 80286 Mode)

Purpose: The .286C pseudo-op tells the Macro Assembler to recognize and assemble 80286 instructions.

Format: .286C

Remarks: The .286C does not have an operand. This mode can be terminated by issuing the .8086 pseudo-op.

Notes:

1. See "Mode Pseudo-Ops" in Chapter 1 for general information.
2. This pseudo-op is not supported by the Small Assembler (ASM).

.8086
(Reset 80286 Mode)

Purpose: The .8086 pseudo-op tells the Macro Assembler not to recognize and assemble 80286 instructions.

Format: .8086

Remarks: The .8086 does not have an operand. This pseudo-op terminates the Macro Assembler 80286 assembler mode.

Notes:

1. The .8086 pseudo-op does not terminate the Macro Assembler 8087/80287 mode.
2. See "Mode Pseudo-Ops" in Chapter 1 for general information.
3. This pseudo-op is not supported by the Small Assembler (ASM).

.8087
(Set 8087-80287 Mode)

Purpose: The .8087 pseudo-op tells the Macro Assembler to recognize and assemble 8087 instructions and data formats.

Format: .8087

Remarks: The .8087 does not have an operand. The .8087 pseudo-op has the same effect on the Macro Assembler as issuing the /R parameter on the DOS command line when invoking the Macro Assembler.

Notes:

1. See "Mode Pseudo-Ops" in Chapter 1 for general information.
2. This pseudo-op is not supported by the Small Assembler (ASM).

Special Macro Operator

Purpose: The ampersand (&) in a macro expansion concatenates text or symbols.

> **Note:** This macro operator is not supported by the Small Assembler (ASM).

Format: *text&text*

Remarks: A dummy parameter that is in a quoted string or not preceded by a delimiter is not substituted in the expansion unless it is immediately preceded by the ampersand (&). To form a symbol from text and a dummy, put the ampersand between them.

Example:

```
ERRGEN     MACRO   X
ERROR&X:   PUSH    BX
ABX        MOV     BX,"&X"
AB&X       JMP     ERROR
           ENDM
```

In this example, the call ERRGEN A generates:

```
ERRORA:   PUSH    BX
ABX       MOV     BX,"A"
ABA       JMP     ERROR
```

;;

Special Macro Operator

Purpose: In a MACRO or repeat, a comment preceded by two semicolons (;;) is not produced as part of the expansion.

> **Note:** This macro operator is not supported by the Small Assembler (ASM).

Format: ;;*text*

Remarks: When two semicolons (;;) are used, the comment does not appear on the listing even when the .LALL pseudo-op is used. However, a comment preceded by one semicolon (;) is preserved and appears in the expansion.

> **Note:** Under the default option of .XALL, a single semicolon remark is not printed either, because it does not generate any machine code.

The double semicolon can significantly reduce the amount of memory workspace used by the definition of a macro. As a macro definition is read, a single semicolon comment is kept in memory, whereas the double semicolon comment is immediately discarded.

The double semicolon comment can be used on a line by itself, or can be used on the comment fields of individual instructions or other statements.

!

Special Macro Operator

Purpose: When an exclamation point (!) is used in an operand, the next character is entered literally.

> **Note:** This macro operator is not supported by the Small Assembler (ASM).

Format: !*character*

Remarks: Equivalent entries are !; and <;>.

%

Special Macro Operator

Purpose: The % special macro operator converts the expression that follows (usually a symbol) to a number in the current RADIX. During macro expansion, the number derived from converting the expression is substituted for the dummy.

> **Note:** This macro operator is not supported by the Small Assembler (ASM).

Format: %*expression*

Remarks: The percent sign (%) is used only in a macro operand. Using the % special operator allows a macro call by value. Usually, a macro call is a call by reference with the text of the macro operand substituting exactly for the dummy.

Example:

```
MAKERR   MACRO    X
LB       =        0
         REPT     X
LB       =        LB+1
         MAKLIB   %LB
         ENDM     ;;END OF REPT
         ENDM     ;;END OF MACRO

MAKLIB   MACRO    Y
ERR&Y:   DB       'ERROR  &Y',0
         ENDM

         MAKERR 3
+ERR1:   DB        'ERROR 1',0
+ERR2:   DB        'ERROR 2',0
+ERR3:   DB        'ERROR 3',0
```

ASSUME

Purpose: ASSUME tells the assembler which segment register a segment belongs to. When the assembler encounters a *seg-name*, it automatically assembles the *seg-name* reference under the proper segment register. ASSUME NOTHING cancels any previous ASSUMEs for the indicated register.

Format:

ASSUME *seg-reg*: *seg-name*[,...]
or
ASSUME NOTHING

Remarks: The valid *seg-reg* entries are:CS, DS, ES, and SS.

The possible entries for *seg-name* are:

- The name of a segment declared with the SEGMENT pseudo-op
- The name of a group declared with the GROUP pseudo-op
- An expression, either:

  `SEG variable-name` or `SEG label-name`

- The keyword NOTHING

  Note: If ASSUME is not used or if the NOTHING is entered for *seg-name*, each *seg-name* type reference must be prefixed by a segment register override.

If a *seg-reg* is to be temporarily used to contain a value other than the *seg-id* as identified in the ASSUME statement, then the ASSUME NOTHING should be used to indicate that the *seg-reg* no longer has that old value. This redefined *seg-reg* can now be used in an explicit reference. A *seg-reg* should never have a value that contradicts what the ASSUME says it has when the contents of the *seg-reg* are being used for addressability.

Example:

```
ASSUME DS:DATA,SS:DATA,CS:CGROUP,ES:NOTHING
 MOV   SI,OFFSET SOURCE
 MOV   DI,OFFSET DEST
 MOV   CX,LENGTH SOURCE
REP MOVS DEST,ES:SOURCE ;ES IS REG BASE OF
;                        BOTH SOURCE AND DEST
```

COMMENT

Purpose: COMMENT lets you enter comments about your program without having to enter semicolons (;) for each line.

Remarks: The first non-blank character encountered after COMMENT is the first delimiter.

Format: COMMENT *delimiter text delimiter*

Remarks: You can enter your comments *text*, and then end your comment block with the same character for the second delimiter. A COMMENT defined in the body of a macro is not displayed unless .LALL is requested.

> Note: The COMMENT pseudo-op should not be used in Structured Assembler source to be processed by the SALUT utility.

Example:

```
COMMENT *You can enter as many lines
of text between the delimiters
o
o
o
as you need to describe your program.*
```

.CREF .XCREF

Purpose: The output of the cross reference information is controlled by these pseudo-ops. The default condition is the .CREF pseudo-op. When a .XCREF pseudo-op is encountered, cross reference information results in no output until .CREF is encountered.

Format: .CREF
or
.XCREF [*operand*]

Remarks: The cross reference utility must be invoked to allow these pseudo-ops to function.

> Note: See Chapter 6 "Cross Reference Utility (CREF)" of the *IBM Personal Computer Macro Assembler* book.

The .XCREF pseudo-op can have an optional operand consisting of a list of one or more variable names that are to be suppressed from appearing in the cross reference listing.

DB
Define Byte

Purpose: DB defines a variable or initializes storage. DB allocates 1 or more bytes (8 bits).

Format: [*variable-name*] DB *expression*[,...]

Remarks: If the optional *variable-name* is entered, the DB pseudo-op defines *variable-name* as a variable. The *variable-name* must not be followed by a colon (:). If the *variable-name* is followed by a colon (:), it is no longer a *variable-name* but is considered to be a code-relative label with the attribute of NEAR, and only addressable in the code (CS) segment. This allows a label to be put on an instruction that is generated in hex. For example:

```
RETRN:   DB   OC3H
```

The *expression* entry can be one of the following:

- A constant expression.
- The question mark (?) for indeterminate initialization.
- A character string of one or more characters (the ASCII characters occupy succeeding character positions).
- The number of constants is limited only by the length of the line. Use a comma to separate the elements. Each element must have a value of 255 or less.
- A duplicated clause.

```
repeat count DUP(expression)...
```

DB
Define Byte

Note: An expression in the form of *repeat count* DUP(?) when used as the only operand of a DB, DW, DD, DT, or DQ definition statement produces an uninitialized data block, suitable for references to PROM-resident locations, which cannot or should not be overwritten by the loader. Any other form of ?, initializes the data block with data of an indeterminate value.

Example:

```
NUM_BASE     DB 16
FILLER       DB  ?
;   INITIALIZE WITH INDETERMINATE VALUE
ONE_CHAR     DB 'B'
MULT_CHAR    DB 'JENNIFER JEFFREY AMARYLLES TOM'
MSG          DB 'MSGTEST',13,10
;   MESSAGE,CARRIAGE RET, & LINEFEED
BUFFER       DB 10 DUP(?)
TABLE        DB 100 DUP(5 DUP(4),7)
;   100 COPIES OF BYTES=4,4,4,4,4,7
NEW_PAGE     DB OCH
;   PRINTER CONTROL
ARRAY        DB 1,2,3,4,5,6,7
```

DD
Define Doubleword

Purpose: Use the DD pseudo-op to define a variable or to initialize storage. DD allocates two words (4 bytes).

Format: [*variable-name*] DD *expression*[,...]

Remarks: If the optional *variable-name* is entered, the DD pseudo-op defines *variable-name* as a variable. The *variable-name* must not be followed by a colon (:). If the *variable-name* is followed by a colon (:), it is considered to be a code-relative label with the attribute of NEAR, and only addressable in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant.
  - Integers are generated, unless a decimal point or scientific notation is used, then floating point is generated.

    Note: Refer to the assembler parameters /R and /E for their effect on floating-point constants.

  - Small Assembler (ASM) expands numeric integer expressions as if DW had been used, then adds 2 zero bytes after the generated integer.
- A question mark (?) for indeterminate initialization.
- An address expression, which generates a 16-bit offset, then the 16-bit Segment base value of the symbol. (Vector)

- The number of constants is limited only by the length of the line. Use a comma to separate the elements.

- A duplicated clause.

```
repeat-count DUP(expression)...
```

Note: An expression in the form of *repeat count* DUP(?) when used as the only operand of a DB, DW, DD, DT, DQ definition statement produces an uninitialized data block, suitable for references to PROM-resident locations, which cannot or should not be overwritten by the loader. Any other form of ?, initializes the data block with data of an indeterminate value.

Example:

```
DBPTR        DD  TABLE       ;16-bit OFFSET,
;                             THEN 16-bit SEG
;                             BASE VALUE
SEC_PER_DAY  DD  60*60*24    ;ARITHMETIC IS
;                             PERFORMED BY
;                             ASSEMBLER
LIST         DD  2 DUP(?)
HIGH         DD  4294967295  ;MAXIMUM
LOW          DD  -4294967295 ;MINIMUM
PI           DD  3.14159     ;FLOATING POINT
;                             SINGLE PRECISION
EPSILON      DD  1.0E-7      ;FLOATING POINT
;                             SCIENTIFIC NOTATION
```

DQ
Define Quadword

Purpose: Use the DQ pseudo-op to define a variable or to initialize storage. DQ allocates four words.

Format: [*variable-name*] DQ *expression*[,...]

Remarks: If the optional *variable-name* is entered, the DQ pseudo-op defines *variable-name* as a variable. The *variable-name* must not be followed by a colon (:). If the *variable-name* is followed by a colon (:), it is considered to be a code-relative label with the attribute of NEAR, and only addressable in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant.
  - Integers are generated, unless a decimal point or scientific notation is used, then floating point is generated.

    Note: Refer to the assembler parameters /R and /E for their effect on floating-point constants.

  - The number of constants is limited only by the length of the line. Use a comma to separate each element.
  - Small Assembler (ASM) expands numeric integer expressions as if DW had been used, then adds 6 zero bytes after the generated integer.
- A question mark (?) for indeterminate initialization.

- A duplicated clause.

```
repeat-count DUPexpression
```

Note: An expression in the form of *repeat count* DUP(?) when used as the only operand of a DB, DW, DD, DT, or DQ definition statement produces an uninitialized data block, suitable for references to PROM-resident locations, which cannot or should not be overwritten by the loader. Any other form of ?, initializes the data block with data of an indeterminate value.

The *expression* entry cannot be:

- An external reference
- A variable
- A label

Example:

```
LONG_REAL   DQ  3.14159265 ;DECIMAL MAKES IT REAL
STRING      DQ  'AB'       ;NO MORE THAN 2 CHARS
HIGH        DQ  18446744073709551615    ;MAXIMUM
LOW         DQ -18446744073709551615    ;MINIMUM
SPACER      DQ  2 DUP(?)      ;UNINITIALIZED DATA
FILLER      DQ  1 DUP(?,?)    ;INITIALIZE WITH
;                              INDETERMINATE VALUE
HEX_REAL    DQ  0FDCBA9A98765432105R
PI          DQ  3.14159       ;FLOATING POINT
;                              DOUBLE PRECISION
EPSILON     DQ  1.0E-14       ;FLOATING POINT
;                              SCIENTIFIC NOTATION
```

DT
Define Tenbytes

Purpose: Use the DT pseudo-op to define a variable or to initialize storage. DT generates 10 bytes of packed decimal.

Format: [*variable-name*] DT *expression*[,...]

Remarks: If the optional *variable-name* is entered, the DT pseudo-op defines *variable-name* as a variable. The *variable-name* must not be followed by a colon (:). If the *variable-name* is followed by a colon (:), it is considered to be a code-relative label with the attribute of NEAR, and only addressable in the code segment (CS).

The *expression* entry can be one of the following:

- A constant.
  - Packed decimal is generated, unless a decimal point or scientific notation is used, then 10 byte temporary format floating-point is generated.

    Note: Refer to the assembler parameters /R and /E for their effect on floating-point constants.

  - Small Assembler (ASM) expands numeric integer expressions as if DW had been used, then adds 8 zero bytes after the generated integer.
- A question mark (?) for indeterminate initialization.

DT
Define Tenbytes

- The number of constants is limited only by the length of the line. Use a comma to separate each element.
- A duplicated clause.

  ```
  repeat-count DUP (expression)...
  ```

  Note: An expression in the form of *repeat count* DUP(?) when used as the only operand of a DB, DW, DD, DT, or DQ definition statement produces an uninitialized data block, suitable for references to PROM-resident locations, which cannot or should not be overwritten by the loader. Any other form of ?, initializes the data block with data of an indeterminate value.

The *expression* entry cannot be:

- An external reference
- A variable
- A label

Example:

```
ACCUMULATOR     DT  ?
STRING          DT  'CD'    ;NO MORE THAN
;                            2 CHARACTERS
PACKED_DECIMAL  DT  1234567890
TEMP_REAL       DT  3.14159 ;FLOATING POINT
;                            TEMP REAL FORMAT
EPSILON         DQ  1.0E-16 ;FLOATING POINT
;                            SCIENTIFIC NOTATION
```

DW
Define Word

Purpose: Use the DW pseudo-op to define a variable or to initialize storage. DW allocates one word (2 bytes).

Format: [*variable-name*] DW *expression*[,...]

Remarks: If the optional *variable-name* is entered, the DW pseudo-op defines *variable-name* as a variable. The *variable-name* must not be followed by a colon (:). If the *variable-name* is followed by a colon (:), it is considered to be a code-relative label with the attribute of NEAR, and only addressable in the code (CS) segment.

The *expression* entry can be one of the following:

- A constant.

  Note: The number of constants is limited only by the length of the line. Use a comma to separate the elements.

- A question mark (?) for indeterminate initialization.
- An address expression.
- A duplicated clause.

```
repeat-count DUP (expression)...
```

Note: An expression in the form of *repeat count* DUP(?) when used as the only operand of a DB, DW, DD, DT, or DQ definition statement produces an uninitialized data block, suitable for references to PROM-resident locations, which cannot or should not be overwritten by the loader. Any other form of ?, initializes the data block with data of an indeterminate value.

Example:

```
ITEMS    DW  TABLE,TABLE+10,TABLE+20  ;OFFSETS
SEGVAL   DW  OFFF0H
BSIZE    DW  4 * 128
LOCATION DW  TOTAL + 1
AREA     DW  100 DUP(?)
CLEARED  DW  50 DUP(0)
SERIES   DW  2 DUP(2,3 DUP(1))  ;2,1,1,1,2,1,1,1
DISTANCE DW  START_TAB - END_TAB
;               DIFFERENCE OF LABELS IS CONSTANT
```

ELSE

Purpose: Each conditional pseudo-op can optionally be used with the ELSE pseudo-op. This allows alternate code to be generated when the opposite condition exists.

Format: ELSE

Remarks: Only one ELSE is permitted for a given IF. A conditional pseudo-op with more than one ELSE or an ELSE without a conditional pseudo-op causes an error. ELSE does not have an operand.

> **Note:** The conditional pseudo-ops can be nested to any level. They are not limited to usage within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation.

Example: All conditional pseudo-ops use the format:

```
IFxx operand
o
o
o
[ELSE] (optional)
o
o
o
ENDIF
```

END

Purpose: The END pseudo-op has two functions:

1. END identifies the end of the source program.
2. The *expression* on the END pseudo-op, when used, identifies the symbol that is the name of the entry point (starting address).

Format: END *[expression]*

Remarks: All source files must have the END as the last statement. Any lines following the END statement are not read by the assembler and are ignored.

When the Linker builds an application program from one or more object modules, it needs to know where the entry point is for DOS to pass initial control. There can only be one DOS entry point for the application. Only one module can identify a label as the entry point by specifying that label on its END statement. Other modules, not having a DOS entry point, must not have an entry point identified on their END statements. If no entry point is identified on any END statement, the Linker sets the entry point to be the beginning of the first linked module. (See the IBM Personal Computer *Disk Operating System* manual for a detailed description of the Linker.)

> **Note:** For applications intended to be used as a ".COM" file, the entry point must be at 100H. This entry point must be identified on the END statement of the first module in the list of linked .OBJ modules, otherwise an error occurs during the EXE2BIN conversion process.

For example:
Observe the usage of the END statements in the two sample source modules, TRYCOM1M and TRYCOM2M, on the assembler diskette.

Example:

```
END BEGIN
```

ENDIF

Purpose: ENDIF ends the corresponding IF conditional pseudo-op. Each IF conditional pseudo-op must be ended by a matching ENDIF conditional pseudo-op.

Format: ENDIF

Remarks: If the IF conditional pseudo-op is not ended by an ENDIF, an "un-terminated conditional" message is generated at the end of each pass of the assembler. An ENDIF without a matching IF causes an error. ENDIF does not have an operand.

> Note: The conditional pseudo-ops can be nested to any level. They are not limited to usage within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation.

Example: All conditional pseudo-ops use the format:

```
IFxx operand
o
o
o
[ELSE] (optional)
o
o
o
ENDIF
```

ENDM

Purpose: Every MACRO, REPT, IRP, and IRPC pseudo-op must be terminated with the ENDM pseudo-op.

> **Note:** The ENDM pseudo-op is not supported by the Small Assembler (ASM).

Format: ENDM

Remarks: If the ENDM pseudo-op is not used with the MACRO, REPT, IRP, and IRPC pseudo-ops, an error is generated. An unmatched ENDM also causes an error.

If the assembler generates an error message stating that it found the end-of-file on the source and cannot find an END statement, when there was an END, the likely cause is a missing ENDM causing the assembler to think the rest of the source is still a part of the MACRO definition.

> **Note:** The *name field* is not allowed. Do not confuse the ENDM pseudo-op with other ending-type pseudo-ops that do require the name of the block being terminated such as ENDP or ENDS.

ENDP

Purpose: Every PROC pseudo-op must be terminated with the ENDP pseudo-op.

Format: *procedure-name* ENDP

Remarks: If the ENDP pseudo-op is not used with the PROC pseudo-ops, an error is generated. An unmatched ENDP also causes an error.

> Note: See the PROC pseudo-op in this chapter for more detail and examples of ENDP use.

Purpose: Every SEGMENT and STRUC pseudo-op must be ended with a corresponding ENDS pseudo-op.

Format: *structure-name* ENDS

or

segname ENDS

Remarks: If the ENDS pseudo-op is not used with the corresponding SEGMENT or STRUC pseudo-op, an error is generated. An unmatched ENDS also causes an error.

> Note: See SEGMENT and STRUC pseudo-ops in this chapter for more details and examples of ENDS use.

EQU

Purpose: The EQU pseudo-op assigns the value of *expression* to *name*.

Format: *name* EQU *expression*

Remarks: If the *name* already has a value other than *expression* or if *expression* is external, a message is generated. Unlike the = (equal sign), the name entry for EQU cannot be redefined to be different from a previous EQU definition of that same symbol.

The EQU pseudo-op cannot be used within a STRUC-ENDS structure definition.

The *expression* entry can be any of the following:

- A symbol
- An index reference
- A segment prefix and operands
- An instruction name
- A record expression
- A 16-bit absolute constant
- A floating-point value

Example:

```
B    EQU [BP+8]    ;(an index reference)
P8   EQU DS:[BP+8] ;(a seg prefix and operand)
CBD  EQU AAD       ;(an instruction name)
EMP  EQU 6         ;(a constant value)
```

Note: Optionally a TYPE attribute can be assigned to the symbol by using the PTR operator. For example:

```
VECTOR  EQU   DWORD PTR 4
```

=
Equal Sign

Purpose: The = pseudo-op allows you to set and then redefine constants. You can redefine the *label* more than once.

Format: *label = expression*

Remarks: The = pseudo-op is similar to the EQU pseudo-op except you can redefine the *label* without generating any message. However, unlike EQU, the = pseudo-op is limited only to numeric expressions (no text strings) that may also include typing by using the PTR operator.

Example:

```
EMP = 6       ;THIS IS THE SAME AS EMP EQU 6
EMP EQU 6     ;ERROR, EMP CANNOT BE REDEFINED
;              BY EQU
EMP = 7       ;EMP CAN BE REDEFINED
EMP = EMP+1   ;CAN REFER TO ITS PREVIOUS
;              DEFINITION
```

Note: Optionally a TYPE attribute can be assigned to the symbol by using the PTR operator. For example:

```
VECTOR   =    DWORD PTR 4
```

Note: If that value were to be changed by some subsequent =, the typing symbol and PTR should be repeated, or the attribute of the symbol will be changed. For example:

```
VECTOR   =    DWORD PTR VECTOR+4
```

EVEN

Purpose: The EVEN pseudo-op causes the program counter to go to an even boundary (an address that begins a word). This ensures that the code or data that follows is aligned on an even boundary.

Format: EVEN

Remarks: If the program counter is not already at an even boundary, EVEN causes the assembler to add a NOP (no operation) so that the counter reaches an even boundary. An error message is generated if EVEN is used with a byte-aligned segment. If the program counter is already at an even boundary, EVEN does nothing.

Example: Before: PC points to 0019 hex (25 decimal).

```
EVEN
```

After: PC points to 001A hex (26 decimal).

EXITM

Purpose: The EXITM pseudo-op is used when a block contains a pseudo-op that tests for some condition and you want to end the REPT, IRP, IRPC, or MACRO call when the test proves that the remainder of the expansion is not required. When an EXITM is executed, the expansion is exited immediately and any remaining expansion or repetition is not generated.

> **Note:** The EXITM pseudo-op is not supported by the Small Assembler (ASM).

Format: EXITM

Remarks: If the block containing the EXITM is nested within another block, the outer level continues to be expanded; however, in all cases the assembler still checks for an ENDM for the MACRO or repeat pseudo-op and returns an error if ENDM is missing. ENDM and EXITM are not interchangeable.

Example:

```
DSEG    SEGMENT
        .
        .
        .
SYM     =      0
        REPT  16
;;CHECK FOR PARA BOUNDARY
        IF    ($-DSEG) MOD 16  EQ  0
        EXITM  ;;QUIT IF PADDED TO BOUNDARY
        ENDIF
SYM     =      SYM + 1
        DB    SYM ;;GENERATE NUMBERED PADDING
        ENDM
```

EXTRN

Purpose: The EXTRN pseudo-op specifies symbols used in this assembly module whose attributes are defined in another assembly module.

Format: EXTRN *name*:*type*[,...]

Remarks: The symbol in the other assembly module must be declared PUBLIC. If the EXTRN pseudo-op is given within a segment, the assembler assumes that the symbol is located within that segment. If the segment is not known, the EXTRN pseudo-op should be placed outside all segments and either used with an explicit segment prefix or an ASSUME pseudo-op.

The *name* entry is the symbol that is defined in another assembly module. The *type* entry can be BYTE, WORD, DWORD, QWORD, TBYTE, NEAR, FAR, ABS, or a name defined by the EQU pseudo-op.

> **Note:** If the *type* of EXTRN is ABS, the defining module must have defined it as being a constant.

For example:

```
FOO       =     5
PUBLIC    FOO
```

Example:

IN THE SAME SEGMENT:

In Module 1:

```
CSEG    SEGMENT
        PUBLIC TAGN
        o
        o
        o
TAGN:
        o
        o
        o
CSEG    ENDS
```

In Module 2:

```
CSEG    SEGMENT
 EXTRN TAGN:NEAR
        o
        o
        o
        JMP TAGN
CSEG    ENDS
```

IN ANOTHER SEGMENT:

In Module 1:

```
CSEGA   SEGMENT
        PUBLIC   TAGF
        o
        o
        o
TAGF:
        o
        o
        o
CSEGA   ENDS
```

In Module 2:

```
 EXTRN TAGF:FAR
CSEGB   SEGMENT
        o
        o
        o
        JMP  TAGF
CSEGB   ENDS
```

GROUP

Purpose: The GROUP pseudo-op collects the segments specified under one name so they are all within a 64K physical segment. (The limit is not checked by the assembler.)

Format: *name* GROUP *seg-name* [,...]

Remarks: The *name* entry is a unique identifier in which the specified *seg-name* entries are to be grouped. The *name* entry cannot be the same name as one of the *seg-name* entries in any GROUP pseudo-op.

The *seg-name* entry can be:

- A segment name assigned by the SEGMENT pseudo-op.
- A SEG *variable* operator.
- A SEG *label* operator.

The order that the segments are named in the *seg-name* entries does not affect the order that the segments are loaded by the Linker. You can specify the order by:

- Using the /A or /S parameters on assembler invocation.

- Specifying the *'class'* entry in the SEGMENT pseudo-op.

- Responding to the "Object module" prompt of the Linker by specifying the object modules in the desired order.

Segment registers can be set once to address all members of the group. (See ASSUME pseudo-op described previously in this chapter.)

Example: Use of the GROUP pseudo-op to combine segments:

In Module A:

```
CGROUP GROUP XXX,YYY
XXX    SEGMENT
       ASSUME CS:CGROUP
       o
       o
       o
XXX    ENDS
YYY    SEGMENT
       ASSUME CS:CGROUP
       o
       o
       o
YYY    ENDS
       END
```

In Module B:

```
CGROUP GROUP ZZZ
ZZZ    SEGMENT
       ASSUME CS:CGROUP
       o
       o
       o
ZZZ    ENDS
       END
```

Uses of the *groupname*:

As immediate:

```
MOV  AX,DGROUP
MOV  DS,AX
```

In assume:

```
ASSUME  DS:DGROUP
```

GROUP

As operand prefix:

```
MOV BX,OFFSET DGROUP:FOO
DW  DGROUP:FOO
DD  DGROUP:FOO
```

This example shows how to set DS with the paragraph number of the group called DGROUP.

Notes:

1. DW FOO returns the offset of the symbol within its segment.
2. DW DGROUP:FOO returns the offset of the symbol within the group.
3. If this DGROUP is defined by Pascal as being the group containing the Pascal Data area, this example produces a MOV immediate into AX of a value that is initialized by the Linker and the Loader to be where the DGROUP WAS, not where the DGROUP IS. Be aware that Pascal is loaded LOW (where the immediate value said it was), but then Pascal during its initialization moves the complete DGROUP up to high storage.

GROUP

This example shows how the GROUP pseudo-op can be used to create a .COM file type.

Example:

```
  PAGE  ,132
  TITLE GRPCOM - USE GROUP TO CREATE .COM FILE
;USE EXE2BIN TO CONVERT GRPCOM.EXE
; TO GRPCOM.COM.
CG  GROUP  CSEG,DSEG ;ALL SEGS IN ON GROUP
DISPLAY MACRO   TEXT
        LOCAL   MSG
DSEG    SEGMENT BYTE PUBLIC 'DATA'
MSG     DB      TEXT,13,10,"$"
DSEG    ENDS
;;MACRO GENERATES PARTLY IN DSEG,
;;PARTLY IN CSEG
        MOV     AH,9
        MOV     DX,OFFSET CG:MSG
;;NOTE USAGE OF GROUP NAME
;;IN GENERATING OFFSET
        INT     21H
        ENDM
DSEG    SEGMENT BYTE PUBLIC 'DATA'
;INSERT LOCAL CONSTANTS AND WORKAREAS HERE
DSEG    ENDS
CSEG    SEGMENT BYTE PUBLIC 'CODE'
  ASSUME CS:CG,DS:CG,SS:CG,ES:CG ;SET BY LOADER
        ORG   100H ;SKIP TO END OF THE PSP
ENTPT   PROC  NEAR ;COM FILE ENTRY AT 100H
     DISPLAY "USING MORE THAN ONE SEGMENT"
     DISPLAY "YET STILL OBEYING .COM RULES"
        RET              ;NEAR RETURN TO DOS
ENTPT ENDP
CSEG  ENDS
      END     ENTPT
```

IFxxxx
Conditional Pseudo-ops

Purpose: Each conditional pseudo-op can optionally be used with the ELSE pseudo-op, which allows alternate code to be generated when the opposite condition exists.

Format: All conditional pseudo-ops use the format:

```
IFxx operand
.
.
.
[ELSE] (optional)
.
.
.
ENDIF
```

Remarks: The conditional pseudo-ops can be nested to any level. They are not limited to usage within a macro. Any operand to a conditional must be known on pass 1 to avoid errors and incorrect evaluation.

IF

Format: IF *expression*

Remarks: True if *expression* is not 0.

IFE

Format: IFE *expression*

Remarks: True if *expression* is 0.

IF1

Format: IF1

Remarks: True if pass 1. IF1 does not have an operand.

> **Note:** See "Pass 1 and Pass 2" in Chapter 5 "Macro Assembler" of the *IBM Personal Computer Macro Assembler* book.

IF2

Format: IF2

Remarks: True if pass 2. IF2 does not have an operand.

> **Note:** See "Pass 1 and Pass 2" in Chapter 5 "Macro Assembler" of the *IBM Personal Computer Macro Assembler* book.

IFDEF

Format: IFDEF *symbol*

Remarks: True if *symbol* is defined or has been declared external via the EXTRN pseudo-op.

> **Note:** The IFDEF pseudo-op is not supported by the Small Assembler (ASM).

IFxxxx Conditional Pseudo-ops

IFNDEF

Format: IFNDEF *symbol*

Remarks: True if *symbol* is undefined or not declared external via the EXTRN pseudo-op.

> **Note:** The IFNDEF pseudo-op is not supported by the Small Assembler (ASM).

IFB

Format: IFB <*operand*>

Remarks: True if *operand* is blank. The angle brackets around <*operand*> are required.

> **Note:** The IFB pseudo-op is not supported by the Small Assembler (ASM).

IFNB

Format: IFNB <*operand*>

Remarks: True if *operand* is not blank. Used for testing when dummy parameters are supplied. The angle brackets around <*operand*> are required.

> **Note:** The IFNB pseudo-op is not supported by the Small Assembler (ASM).

IFIDN

Format: IFIDN <*operand-1*>,<*operand-2*>

Remarks: True if the string *operand-1* is identical to the string *operand-2*. The angle brackets around the operands are required.

> **Note:** The IFIDN pseudo-op is not supported by the Small Assembler (ASM).

IFDIF

Format: IFDIF <*operand-1*>,<*operand-2*>

Remarks: True if the string *operand-1* is different from the string *operand-2*. The angle brackets around the operands are required.

> **Note:** The IFDIF pseudo-op is not supported by the Small Assembler (ASM).

ENDIF

Format: ENDIF

Remarks: Each IF pseudo-op must have a matching ENDIF to end the conditional; otherwise, an "un-terminated conditional" message is generated at the end of each pass of the assembler. An ENDIF without a matching IF causes an error. ENDIF does not have an operand.

IFxxxx
Conditional Pseudo-ops

ELSE

Format: ELSE

Remarks: Each conditional pseudo-op can optionally be used with the ELSE pseudo-op, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF. A conditional pseudo-op with more than one ELSE or an ELSE without a conditional pseudo-op causes an error. ELSE does not have an operand.

Example: All conditional pseudo-ops use the format:

```
IFxx operand
o
o
o
[ELSE] (optional)
o
o
o
ENDIF
```

INCLUDE

Purpose: The INCLUDE pseudo-op assembles your source statements from an alternate source file into the current source file. Use of the INCLUDE pseudo-op eliminates the need to repeat a sequence of statements that are common to several source files.

Format: INCLUDE [drive][path]*filename*[.ext]

Remarks: The *filespec* entry can be any valid file specification as determined by the Disk Operating System.

The INCLUDE file is opened and assembled into the current source file immediately following the INCLUDE pseudo-op statement. When the end-of-file is reached, assembly resumes with the statement following the INCLUDE.

A "C" is printed between the assembled code and the source line in column 30 of each line that is assembled from an INCLUDE file. Column 32 of the listing is a space. Column 33 is column 1 of the source.

The file specified in the *filespec* entry must exist; otherwise, a message results and the INCLUDE pseudo-op is ignored.

Nested INCLUDES are allowed; however, each open INCLUDE file requires an additional 650 bytes of storage from available work space. In the Small Assembler (ASM), which has a minimum of work space anyway, nesting INCLUDEs can easily result in an "Out of memory" error when used in a 64K machine.

INCLUDE

When nesting INCLUDE files, using IBM Personal Computer DOS version 2.0 and later, the CONFIG.SYS file needs to have added a FILES=*n* parameter, where *n* is greater than the default of 8. Otherwise, if there is any nesting of INCLUDEs, then all three output files normally cannot be produced on one run of MASM. For each level of nesting of INCLUDE, one more output file must be suppressed. IBM Personal Computer DOS version 2.0 and later has a default of a maximum of only 3 simultaneously open files. When MASM tries to nest INCLUDEs, that limit is exceeded and an open error results.

Example:

```
IF1
INCLUDE B:MYMACR.LIB
ENDIF
```

IRP

Purpose: The number of operands in the <*operandlist*> entry determines the number of times the block of statements are repeated. Each repetition substitutes the next item in the <*operandlist*> entry for every occurrence of *dummy* in the block.

> **Note:** The IRP pseudo-op is not supported by the Small Assembler (ASM).

Format: IRP *dummy*,<*operandlist*>

```
    .
    .
    .
    ENDM
```

Remarks: The <*operandlist*> entry must be enclosed in angle brackets. If the <*operandlist*> is null (<>), the block is processed once with each occurrence of <*dummy*> removed.

The IRP—ENDM block does not necessarily have to be within a macro definition.

Example: In this example, the assembler generates the code DB 1 through DB 10.

```
IRP     X,<1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

In this example:

```
IRP  DUMMY,<"first line",13,10,"second line",13,10>
DB   DUMMY
ENDM
```

The assembler generates the code:

```
DB       "first line"
DB       13
DB       10
DB       "second line"
DB       13
DB       10
```

Purpose: The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of *dummy* in the block.

> **Note:** The IRPC pseudo-op is not supported by the Small Assembler (ASM).

Format: IRPC *dummy,string* (or <*string*>)

```
   .
   .
   .
   ENDM
```

Remarks: The IRPC pseudo-op is similar to the IRP pseudo-op except that <*operandlist*> is replaced by a string of text and the angle brackets around the string are optional.

The IRPC—ENDM block does not necessarily have to be within a macro definition.

Example:

```
IRPC   X,01234567
DB     X+1
ENDM
```

In this example, the assembler generates the code DB 1 through DB 8.

LABEL

Purpose: The LABEL pseudo-op defines the attributes of *name*:

- Segment: current segment being assembled
- Offset: current position within this segment
- Type: specified as an operand

Format: *name* LABEL *type*

Remarks: The *type* entry must be one of the following:

For data areas:

- BYTE
- WORD
- DWORD
- QWORD
- TBYTE
- *structure name*
- *record name*

For executable code:

- NEAR
- FAR

The LABEL pseudo-op cannot be used within a STRUC-ENDS structure definition.

Example: To reference a data area but use a length different from that area's original definition:

```
BARRAY  LABEL  BYTE
ARRAY   DW     100 DUP(0)
        o
        o
        o
        ADD    AL,BARRAY[99]  ;ADD 100th BYTE
;                                      TO AL
        ADD    AX,ARRAY[98]   ;ADD 50th WORD
;                                      TO AX
```

To define multiple entry points within a procedure:

```
SUBRT   PROC    FAR
        o
        o
        o
SUB2    LABEL   FAR ;SHOULD HAVE
;                    SAME ATTRIBUTE
;                    AS CONTAINING
;                    PROC
        o
        o
        o
        RET
SUBRT   ENDP
```

.LALL .SALL .XALL

Purpose: The .LALL lists the complete macro text for all expansions. .LALL also allows for the optional display of false conditional blocks by using the pseudo-ops .TFCOND, .LFCOND, .SFCOND or the assembler command line parameter /X.

The .SALL suppresses listing of all text and object code produced by the macros.

The .XALL is the default condition; a source line is listed only if it generates object code.

Format:

```
.LALL
 or
.SALL
 or
.XALL
```

.LFCOND
(List False Conditionals)

Purpose:	.LFCOND (List False Conditionals) causes the listing of conditional blocks that evaluate as false.
Format:	.LFCOND
Remarks:	.LFCOND does not have an operand. This state can be terminated either by issuing .TFCOND (which reverts to the default state concerning listing of false conditionals, but with the default state redefined as being in the opposite state) or by issuing the .SFCOND, which halts the listing of false conditionals.

False conditionals within macros are not printed when in the .LALL state.

Note: See "False Conditional Blocks" in Chapter 1 for detail.

.LIST .XLIST

Purpose: Output to the listing file can be controlled by these two pseudo-ops.

Format:

.LIST
or
.XLIST

Remarks: If a listing is not being made, these pseudo-ops have no effect on the listing. The .LIST is the default condition. When a .XLIST is encountered, the source and the object code is not listed until a .LIST pseudo-op is encountered.

The /D parameter causes the .XLIST to be ignored.

LOCAL

Purpose: When LOCAL is executed, the assembler creates a unique symbol for each *dummy* entry in *dummylist* and substitutes that symbol for each occurrence of the *dummy* in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro.

> **Note:** The LOCAL pseudo-op is not supported by the Small Assembler (ASM).

Format: LOCAL *dummylist*

Remarks: The LOCAL pseudo-op is allowed only inside a MACRO pseudo-op. The symbols created by the assembler range from ??0001 to ??FFFF. You must avoid using the form ??nnnn for your own symbols. If LOCAL statements are used, they must be the first statements in the MACRO pseudo-op. Not even COMMENT or semicolon remarks can be placed between MACRO and LOCAL.

Multiple LOCAL statements can be used, if the *dummylist* is too long to fit on one line, or there is a desire for a vertical list of LOCAL symbols.

Example:

```
DISPLAY MACRO   TT
        LOCAL   AGAIN
;; DISPLAY MSG POINTED TO BY BX TT TIMES
        MOV     CX,TT
        MOV     AH,9
        MOV     DX,BX
AGAIN:  INT     21H
        LOOP    AGAIN
        ENDM
```

MACRO

Purpose: Often it is convenient to be able to generate a given sequence of statements from various places in your program even though different parameters may be required each time the sequence is invoked. This capability is provided by the MACRO pseudo-op.

Note: The MACRO pseudo-op is not supported by the Small Assembler (ASM).

Format: *name* MACRO *dummylist*

```
     .
     .
     .
     ENDM
```

Remarks: The *name* entry can be the same as a pseudo-op or instruction. If *name* matches an instruction mnemonic or assembler pseudo-op (except TITLE), the macro definition is used, replacing the normal meaning of the symbol. PURGE can be used to delete the macro. The *name* entry is used to invoke the macro.

A macro consists of three essential parts:

1. The MACRO pseudo-op, defining the *name* and *dummylist*.
2. The body of the macro, containing the prototypes of statements to be generated when the macro's name is invoked.
3. The ENDM pseudo-op to terminate the definition of the macro.

The invocation of a macro has no name field, but specifies the macro *name* (defined in the *name* field of the MACRO definition statement) as a pseudo-op followed, after at least one blank, with the *parmlist*. For example:

name parmlist

Each time the macro's *name* is invoked, the statements in the body of the macro are generated and flagged in the assembler's listing with a plus (+) symbol two print positions to the left of where column one of the source is printed.

The *dummylist* is a series of one or more symbols, separated by a comma (no blanks), defining the symbols that are to be replaced when referenced within the body of the macro. When the macro is expanded (generated), each usage of a symbol defined in the *dummylist* is replaced by an entry in the *parmlist*, specified on the macro invocation statement. The *parmlist* is also a series of elements, separated by a comma.

The relative positions of the elements are important, since the substitution is done on a one-to-one basis. That is, the first symbol defined on the *dummylist* is replaced by the first element defined on the invocation statement's *parmlist*; the second symbol by the second element, and so on for the entire list. The substitution is a replacement of the dummy symbol with the character string specified as the corresponding parameter. The number of parameters used when the macro is called need not be the same as the number of *dummylist* entries. If you have more parameters than *dummylist* entries, the extra ones are ignored. If fewer, the extra *dummylist* entries become nulls. If the parameter has a prefix of "%" in the *parmlist*, the dummy is substituted with the value of the parameter instead of the character string of the parameter itself.

Because a blank after an element in the *parmlist* will be treated as a specification of a null value for the next positional parameter, blanks should not appear in the *parmlist*. If it is desired for a parameter to contain a significant blank or comma, the inclusion of that special character as part of the parameter can be indicated by enclosing the entire parameter element within angle brackets. For example:

```
PUSHVEC MACRO   PARM1,PARM2
        MOV     AX,PARM1
        PUSH    AX
        MOV     AX,PARM2
        PUSH    AX
        ENDM
        o
        o
        o
        PUSHVEC DS,<OFFSET VARNAME>
;PUSH DWORD VECTOR OF VARNAME ONTO STACK
```

This usage of the angle brackets can also be used to generate a variable length of generated results. For example:

```
STRING  MACRO   NUMBERS
        DB      NUMBERS
        ENDM
        o
        o
        o
        STRING  <1,2,3,4>
;GENERATE 4 BYTES OF INTEGER NUMBERS
```

Notes:

1. A dummy parameter in a MACRO is always recognized exclusively as a dummy parameter. Register names such as AL and BX are changed in the expansion if they were used as dummy parameters. The replacement of dummy parameters does not occur in comments.

2. See Chapter 1 for additional information about macros.

Example:

```
GEN     MACRO     XX,YY,ZZ
        MOV       AX,XX
        ADD       AX,YY
        MOV       ZZ,AX
        ENDM
```

When the call is invoked, for example:

```
        GEN       ED,KISER,SUM
```

The assembler generates the following code:

```
        MOV       AX,ED
        ADD       AX,KISER
        MOV       SUM,AX
```

NAME

Purpose: To give a module a name.

Format: NAME *module-name*

Remarks: The *module-name* cannot be a reserved word. This pseudo-op can be labeled.

Every module is named. The assembler selects the name from the following list in the order shown:

1. NAME pseudo-op (only one NAME pseudo-op allowed per assembly).
2. If NAME pseudo-op is not present, the first six characters of text on a TITLE statement, if the six characters are valid for use in the name field.
3. If TITLE statement is not present or the first six characters are not valid for use in a name field, then the source filename becomes the module name.

The *module-name* is passed to the Linker and Library Manager.

ORG

Purpose: The location counter is set to the value of *expression*. The assembler assigns generated code starting with the value.

Format: ORG *expression*

Remarks: All names used in *expression* must be known on pass 1, and the value must be either absolute or in the same area as the location counter.

The *expression* must evaluate to a 2-byte absolute number.

The dollar sign ($) can be used to refer to the current value of the location counter.

The ORG pseudo-op cannot be used within a STRUC-ENDS structure definition.

Example:

```
ORG  120H
ORG  $+2        ;SKIP NEXT 2 BYTES
```

To conditionally skip to the next 256-byte boundary:

```
CSEG   SEGMENT PAGE
BEGIN  = $
       o
       o
       o
       IF ($-BEGIN) MOD 256
;IF NOT ALREADY ON 256 BYTE BOUNDARY
       ORG ($-BEGIN)+256-(($-BEGIN) MOD 256)
       ENDIF
```

%OUT

Purpose: The %OUT pseudo-op displays progress through a long assembly or displays the value of conditional assembly parameters.

Format: %OUT *text*

Remarks: The *text* entry is listed on the display during assembly when the %OUT is encountered.

> Note: The %OUT text is sent to the console even if the output list file is also to the console via USER or CON. The use of USER is preferred rather than CON.

Example: 1

```
        IF   IBM
        %OUT IBM VERSION
        ENDIF

        IF2
        %OUT STARTING SECOND PASS...
        ENDIF
```

Example: 2

```
INNER   MACRO   TEXT,VAL
        %OUT    TEXT  VAL
        ENDM
        o
        o
        o
HERE    =       $ - CSEG
        INNER   <CURRENT LOCATION>,%HERE
```

PAGE

Purpose: The PAGE pseudo-op controls the length and width of each listing page. The PAGE pseudo-op is placed in the source file to control the formatting and the printing of the listing file generated during assembly.

Format: PAGE [*operand-1*][*operand-2*]

Remarks: Using the PAGE pseudo-op without the operand entries causes the printer to go to the top of the page and increments the page number by 1.

Each page of the listing produced by the assembler contains a chapter number and a page number separated by a dash. The page number is incremented when a page is full or when PAGE (no operand) is encountered. The chapter number is incremented only when PAGE + is encountered. Both cause the printer to go to the top of the next page.

The *operand-1* entry can be:

- A number within the range of 10 to 255 indicating the number of lines-printed-per-page (length). The page length without a specified number is 58.

  Note: The assembler causes a printer eject character to be printed on the 58th line of each page if a number is not specified. This setting allows a margin at the top and bottom of each page with the standard 66 line-per-page paper.

- The + sign indicating that the chapter number is to be incremented by 1 and the page number set to 1. If you use the + sign, it cannot be used with any other operand.

Use the *operand-2* entry to control the width of the page. The page width without a specified number is 80. *Operand-2* can be specified in the range of 60 to 132.

> **Note:** The PAGE pseudo-op does not set the printer to the desired line width. For proper formatting of the listing, initialize the printer to operate at a corresponding line width. Do this before printing the listing file by some other method, such as the DOS MODE command.

See "MODE command" in the IBM Personal Computer *Disk Operating System* (DOS) manual to properly set the printer. For example:

```
MODE LPT1:132
```

can be used at the DOS prompt to set the IBM Personal Computer Printer to 132 columns wide.

Another method is to imbed the printer control characters that set the character size into the comment field of the PAGE statement (see the SKELEXE.ASM example file on the Macro Assembler diskette).

Purpose: The PROC pseudo-op identifies a block of code. By dividing the code into blocks, each of which performs a distinct function, the overall function of the complete module can be made clearer for a better understanding.

The PROC pseudo-op also identifies the type of linkage to be used by any RET instruction contained within the block of code.

Format:

procedure-name PROC [NEAR]
or
procedure-name PROC FAR
∘
∘
∘
RET
∘
∘
∘
procedure-name ENDP

Remarks: The block of code identified by the PROC pseudo-op can be executed in-line, jumped to, or started by a CALL. For the *procedure-name* to be accessible to any external CALL or other external reference, you must also use the PUBLIC pseudo-op. If the PROC is the entry point of an .EXE module that is to be invoked directly by the DOS loader, code the PROC attribute as FAR. If the PROC is called from code that has another ASSUME CS value, the FAR attribute must be used.

The NEAR specification causes any RET coded within the procedure to be an intra-segment return that pops a return offset from the stack. A NEAR subroutine can only be called from the same segment. The FAR, however, causes RET to be an inter-segment return that pops both a return offset and a segment base from the stack. A FAR subroutine can be called from any segment, and is usually called from a segment other than the one containing the subroutine.

Example: In this example, the lower subroutine is called by the upper subroutine.

```
          PUBLIC  FAR_NAME
FAR_NAME  PROC    FAR
          CALL    NEAR_NAME
          RET  ;POPS RETURN OFFSET AND SEG ID
FAR_NAME  ENDP


           PUBLIC  NEAR_NAME
NEAR_NAME  PROC    NEAR
           o
           o
           o
           RET     ;POPS ONLY RET OFFSET
NEAR_NAME  ENDP
```

The lower subroutine can be called directly from a NEAR segment by using:

```
CALL NEAR_NAME
```

A FAR segment can indirectly invoke the second subroutine by first calling the upper subroutine with:

```
CALL FAR_NAME
```

A CALL to a forward-referenced symbol will assume the symbol is NEAR. If that symbol is FAR, the CALL must have an override, for example:

```
CALL FAR PTR FORREF
```

PUBLIC

Purpose: The PUBLIC pseudo-op makes the defined symbols available for use by other programs that are to be concurrently linked. The information referenced by the PUBLIC pseudo-op is passed to the Linker.

Format: PUBLIC *symbol*[,...]

Remarks: The *symbol* entry can be a number, a variable, or a label (including PROC labels). Register names and any symbols defined by EQU or = to floating-point numbers or integers larger than 2 bytes are invalid entries.

Example:

```
         PUBLIC GETINFO
GETINFO  PROC   FAR
         PUSH   BP     ;SAVE CALLER'S REG
         MOV    BP,SP  ;GET ADDR PARMS
                       ;BODY OF SUBROUTINE
         POP    BP     ;RESTORE CALLER'S REG
         RET           ;RETURN TO CALLER
GETINFO  ENDP
```

PURGE

Purpose: The PURGE pseudo-op deletes the definition of a specified MACRO entry allowing space to be reused.

Note: The PURGE pseudo-op is not supported by the Small Assembler (ASM).

Format: PURGE *macro-name*[,...]

Remarks: It is not necessary to PURGE a MACRO before redefining it.

PURGE can relieve the "Out of Memory" condition by deleting the contents of un-referenced macros. This condition can occur if a large, general-purpose macro library is included.

Example:

```
PURGE   MACRONAME
```

performs the same function as redefining a macro with no contents, as in:

```
MACRONAME   MACRO
            ENDM
```

In the following example, assume that MAC1 is one of the macros included in MACRO.LIB.

```
INCLUDE   MACRO.LIB
PURGE     MAC1
MAC1     ;INVOKES THE PURGED MACRO
         ; BUT GENERATES NOTHING
```

Purpose: The .RADIX pseudo-op allows the default RADIX (decimal) to be changed to any base within the range 2 to 16.

Format: .RADIX *expression*

Remarks: The *expression* entry is always in decimal RADIX regardless of the current RADIX.

.RADIX has no effect on DD, DQ or DT. These always assume decimal, unless specifically overridden with a data type suffix. The suffix overrides are:

B — Binary
D — Decimal
H — Hexadecimal Integer
Q or O — Octal
R — Hexadecimal Real

When using .RADIX 16, be aware that if the hex constant ends in either B or D, the assembler thinks that is a request to override the current RADIX specification with a base of BINARY or DECIMAL respectively. In such cases, do not let the hex number end with just a B or D without also adding the H base override just as if .RADIX 16 was not in use. For example:

```
.RADIX   16
DW   120B
```

generates an error, since the 2 is not a valid binary number.

The correct specification is:

```
DW   120BH
```

Another example:

```
.RADIX   16
DW   89CD
```

also generates an error, because the C is not a valid decimal number. The correct specification should be:

```
DW   89CDH
```

The dangerous part is when no error is generated. For example:

```
.RADIX   16
DW   120D
```

generates a constant whose value is one hundred twenty decimal, not '120D' hex, which may have been the intended value.

Example: The following two move instructions are the same:

```
MOV     BX,0FFH
.RADIX  16
MOV     BX,0FF
```

Note: The "H" suffix is still required for hex numbers ending in "B" or "D" regardless of the current .RADIX specification. For example:

```
DW   89ADH
DW   102BH
```

RECORD

Purpose: A record is a bit pattern you define to format bytes and words for bit-packing. The *recordname* itself becomes a pseudo-op used to allocate storage.

> **Note:** The RECORD pseudo-op is not supported by the Small Assembler (ASM).

Format: *recordname* RECORD *fieldname*:*width*[=*exp*],[..]

Remarks: The *recordname* and *fieldname* entries are unique identifiers and must be used. You must enter the colon (:) between the *fieldname* and *width*. The *fieldname* entry is the name of the field. The value of *fieldname*, when used in an expression, is the shift count to move the field to the far right. The MASK operator returns a bit mask for the field.

The *width* entry evaluates to a constant in the range 1 to 16, and specifies the number of bits defined by *fieldname*. If the total width is larger than 8 bits, the assembler uses 2 bytes; otherwise, 1 byte is used. If the total number of bits defined is less than 8 (a byte) or 16 (a word), the fields are right-justified into the least significant bit positions of the byte or word.

The *exp* entry contains the default value for the field. If the field is at least 7 bits wide, it can be initialized to an ASCII character (for example, FIELD:7='Z').

Records can be used in expressions in the form:

```
recordname <[init-list]>
```

The <[*init-list*]> entry is an optional list of the initialization values. The angle brackets must be coded as shown but the values do not have to be given.

To initialize a record use the form:

```
[name] recordname <[exp][,...]>
   or
[name] recordname exp DUP(<[exp]>)
```

The *name* entry is an optional name for the first byte or word of the allocated storage. The *recordname* specifies the name (assigned by you) of the record from the RECORD pseudo-op that defines the format and optional default field values. The <[*exp*][,...]> entry specifies a list of field-initialization or optional override values so that trailing fields default.

In the second form above, the angle brackets (shown outside the square brackets) around the second exp are required. If [*exp*] is left blank, either the default value applies or the value is indeterminate.

Example: Define the record fields, most significant fields first:

```
MODULE RECORD R:7,E:4,D:5
```

Fields are 7 bits, 4 bits, and 5 bits; no default value is given. Shown most significant byte first, this looks like:

```
RRRR RRRE EEED DDDD
```

To allocate its storage:

```
STG_FLD  MODULE  <7,,2>
```

This defines R=7 and D=2, and leaves E indeterminate; it generates 2 bytes, the least significant byte first:

```
02 0E
```

Define the record fields:

```
AREA  RECORD FLA:8='A',FLB:8='B'
```

To allocate its storage:

```
CHAR_FLD AREA <,'P'>
```

This defines FLA='A' (the default) and changes FLB='P'.

Notes:

1. Be aware of the 132-character limit on the length of lines allowed by the assembler when defining a RECORD. The 132-character limit may mean the selection of short names for the fields in order to fit on one line.

2. See the FNSTSW instruction in Chapter 3 for a sample RECORD defining the status bits in the 8087 STATUS WORD.

To use a field:

```
DEFFIELD  RECORD   X:3,Y:4,Z:9
          o
          o
          o
TABLE     DEFFIELD 10 DUP(<0,2,255>)
          o
          o
          o
          MOV DX,TABLE[2]
;MOVE DATA FROM RECORD TO REGISTER
; OTHER THAN SEGMENT REGISTER
          AND DX,MASK Y
;MASK OUT FIELDS X AND Z
; TO REMOVE UNWANTED FIELDS
; The MASK of Y equals 1E00H
; 0001111000000000B (1E00H) IS THE VALUE
          MOV CL,Y    ;GET SHIFT COUNT
;                      9 IS THE VALUE
          SHR DX,CL   ;FIELD TO LOW-ORDER
; BITS OF REGISTER, DX IS NOW EQUAL TO
; THE VALUE OF FIELD Y
          MOV CL,WIDTH Y ;GET NUMBER OF
;                         BITS IN FIELD
;                         4 IS THE VALUE
;The WIDTH of Y equals 4.
```

Purpose: The block of statements between REPT and ENDM is repeated the number of times in the *expression* entry.

> **Note:** The REPT pseudo-op is not supported by the Small Assembler (ASM).

Format:

```
REPT expression
o
o
o
ENDM
```

Remarks: If the *expression* entry contains any external or undefined terms, an error is generated.

The REPT—ENDM block does not necessarily have to be within a macro definition.

Example: This example generates a series of sequenced bytes up to a paragraph boundary.

```
DSEG    SEGMENT
        o
        o
        o
SYM     =     0
        REPT  16
;;CHECK FOR PARA BOUNDARY
        IF    ($-DSEG) MOD 16  EQ  0
        EXITM
        ENDIF
SYM     =     SYM + 1
        DB    SYM
        ENDM
```

SEGMENT

Purpose: At run time, each instruction and each variable of your program lies within some segment. All code generation instructions should be defined as being within a SEGMENT. Your assembly module can be a part of a segment, a segment, parts of several segments, several segments, or a combination of these.

Format: *segname* SEGMENT [*align-type*][*combine-type*] [*'class'*]

.

.

.

segname ENDS

Remarks: The symbol entry for a segment is assigned attributes of an *align-type* entry, a *combine-type*, and a character string defining *'class'* if present and the length of the segment must also be kept. If the segment is named in a GROUP pseudo-op, the segment must include a pointer to the symbol table entry for the GROUP.

> **Note:** If you specify two or three attributes, separate them with a space.

The *align-type* entry can be PARA, BYTE, WORD, or PAGE. The definition of these types are:

- PARA (Default): This align-type specifies that the segment begins on a paragraph boundary (the address is divisible by 16). That is, the least significant hexadecimal digit of the address equals 0H.

- BYTE: This align-type specifies that the segment can begin anywhere.

- WORD: This align-type specifies that the segment begins on a word boundary (an even address where the least significant bit of the address equals 0.

- PAGE: This align-type specifies that the segment begins on a page boundary (an address divisible by 256 and whose two least significant hexadecimal digits of the address are equal to 00H.)

The *combine-type* entry can be PUBLIC, COMMON, AT *exp*, STACK, or no entry (which defaults to not combinable). The definition for these types are:

- PUBLIC: This combine-type specifies that this segment will be concatenated to others of the same name when linked.

- COMMON: This combine-type specifies that this segment and all other segments of the same name that are linked together begin at the same address, and thus overlap. The length of a linked COMMON is the maximum of the linked segments.

- AT *expression*: This combine-type specifies that this segment is to be located at the 16-bit paragraph number evaluated from the given expression. The expression can be any valid constant; however, it cannot be a forward reference. AT *expression* cannot be used to force loading of code at a fixed address; rather, it permits labels or variables to be defined at fixed offsets within fixed areas of storage, such as Read Only Memory or the vector space in low storage. No code is generated for this segment.

- STACK: This combine-type specifies that this segment is to be part of the run-time stack segment, accessed last-in first-out (LIFO) using the assembler instructions such as: PUSH, POP, CALL, INT, IRET, POPF, PUSHF, and RET. If the application is being written to be an .EXE file, a STACK segment is required; if a .COM file, a STACK segment should not be assembled. The Linker issues a warning message if no STACK segment is defined. Ignore that Linker warning message if the application is to be converted from an .EXE file to a .COM file.

- MEMORY: This combine-type, although recognized by the assembler, is not supported by the Linker (LINK), and should not be used. If used, it will be treated as "PUBLIC".

The *'class'* entry is the name (must be enclosed in single quotes) used to group segments at Linker time. Refer to the IBM Personal Computer *Disk Operating System* manual for a description of the Linker.

All three of the above operands are passed to the Linker.

Segment definitions can be nested. When segments are nested, the assembler acts as if they are not and processes them sequentially by appending the second part of the split segment to the first. When the ENDS pseudo-op is detected by the assembler, the assembler takes up the next segment, completes it, and continues. Overlapping segments are not permitted.

Example: **Combining logical segments into a physical segment:**

In module A—

```
SEGA   SEGMENT  PUBLIC
       ASSUME   CS:SEGA
       o
       o
       o
SEGA   ENDS
       END
```

In module B—

```
SEGA  SEGMENT  PUBLIC
      ASSUME   CS:SEGA
      o (Note: Linker adds this
      o  segment to same named
      o  segment in module A)
SEGA  ENDS
      END
```

Nesting of Segments:

```
CSEG  SEGMENT
      o
      o
      o
      MOV  AX,BX
DSEG  SEGMENT
  P1  DW   6
DSEG  ENDS
      o  (CSEG continues ...)
      o
      o
CSEG  ENDS
```

.SFCOND

Purpose: .SFCOND suppresses the listing of conditional blocks that evaluate as false.

Format: .SFCOND

Remarks: .SFCOND does not have an operand. This state can be terminated either by issuing .TFCOND (which reverts to the default state concerning listing of false conditionals, but with the default state redefined as being in the opposite state) or by issuing the .LFCOND, which forces the listing of false conditionals.

> Note: See "False Conditional Blocks" in Chapter 1 for detail.

STRUC

Purpose: The STRUC pseudo-op is like the RECORD pseudo-op except that STRUC has a multi-byte capability. The allocation and initialization of a STRUC block is the same as for the RECORD pseudo-op.

> **Note:** The STRUC pseudo-op is not supported by the Small Assembler (ASM).

Format: *structurename* STRUC

°
°
°

variable-name (a DEFINE pseudo-op) *exp*

°
°
°

structurename ENDS

Remarks: The *structurename* itself becomes a pseudo-op used to allocate storage. Inside the STRUC/ENDS block, DB, DW, DD, DQ, and DT pseudo-ops can be used to allocate space. It is not necessary for a data field defined within a STRUC to have a variable name. Any labels on a DEFINE pseudo-op inside the STRUC/ENDS block become the *fieldnames*. Initial values given in the STRUC/ENDS block are default values for the various fields. These values, or *fields* are of two types:

- overridable
- not overridable

A simple field (a field with only one entry) is overridable. A multiple field (a field with more than one entry) is not overridable.

If the *field* contains a string, it can be overridden by another string. However, if the overriding string is shorter than the initial string, the assembler pads the space to the right with blanks. If the overriding string is longer, the assembler truncates the extra characters.

The format used to reference an item defined within the STRUC/ENDS block (for example, an operand) is:

```
variable.field
```

The *variable* entry represents a variable. The *field* entry represents a label given to a define data pseudo-op inside the STRUC/ENDS block (code the period as shown). The value of the *field* entry is the offset within the addressed structure.

Example:

```
Definition:

STR     STRUC
FIELD1  DB 1,2              ;IS NOT OVERRIDABLE
FIELD2  DB 10 DUP(?)        ;IS NOT OVERRIDABLE
FIELD3  DB 5                ;IS OVERRIDABLE
FIELD4  DB 'BREINER'        ;IS OVERRIDABLE
STR     ENDS
```

Allocation:

```
DB_AREA STR <,,7,'TOM C'> ;OVERRIDES 3rd & 4th
;                          FIELDS ONLY
```

Reference:

```
        MOV AL,[BX].FIELD3
        MOV AL,DB_AREA.FIELD3
```

SUBTTL

Purpose:	The SUBTTL pseudo-op specifies a subtitle to be listed on the line after the title on each page heading.
Format:	SUBTTL *text*
Remarks:	The *text* entry is truncated after 60 characters. Any number of SUBTTLs can be given in a program.

Subtitles are turned off unless the SUBTTL pseudo-op is used. To turn off subtitles again for a portion of the listing after the SUBTTL pseudo-op was given, use another SUBTTL pseudo-op but with a null entry in the *text* string.

SUBTTL does not cause a skip to the top of the page.

SUBTTL is usually to be followed by PAGE.

.TFCOND

Purpose: Toggles the default setting that controls the listing of false conditionals, then sets the current condition to that of the new default, thus terminating the effect of a .SFCOND or .LFCOND previously issued.

Format: .TFCOND

Remarks: .TFCOND does not have an operand. The .TFCOND pseudo-op sets the current and default setting to the non-default condition. If the /X parameter is given in the assembler command line for a file that contains .TFCOND, the /X parameter reverses the effect of the .TFCOND pseudo-op.

Note: See "False Conditional Blocks" in Chapter 1 for detail.

TITLE

Purpose: The TITLE pseudo-op specifies a title to be listed on the second line of each page.

Format: TITLE *text*

Remarks: If more than one TITLE is given, an error results. The first six characters of the title are used as the module name unless a NAME pseudo-op is used. If you do not use a NAME or TITLE pseudo-op, the module name is created from the source filename.

If a TITLE is not given or any of the six characters of the TITLE text are not valid for use in a name field, the TITLE is not used for the name. The name is created from the name of the source file.

The TITLE pseudo-op is placed in the source file to control the formatting and the printing of the listing file generated during assembly.

The *text* entry is limited to 60 characters.

Chapter 3. Instruction Mnemonics

This chapter describes the instructions used by your IBM Personal Computer Macro Assembler.

The instructions are arranged in alphabetic order for ease of reference.

See Chapter 1 for general information about instructions.

AAA
ASCII Adjust for Addition

Purpose: AAA does a correction of the result in AL of adding two unpacked decimal operands, resulting in an unpacked decimal sum.

Format: AAA (no operands)

Remarks: If the lower half-byte (4 bits) of AL is greater than 9 or if the auxiliary carry flag has been set, 6 is added to AL and 1 is added to AH. AF and CF are set. The new value of AL has an upper half-byte of all zeroes, and the lower half-byte is the number between 0 and 9 created by the above addition.

Logic:

```
if ((AL) & 0FH) > 9 or (AF)=1 then
   (AL) ← (AL) + 6
   (AH) ← (AH) + 1
   (AF) ← 1
   (CF) ← (AF)
   (AL) ← (AL) & 0FH
```

Flags: Affected— AF,CF
Undefined— OF,PF,SF,ZF

Encoding: 00110111
37

Example:

```
AAA    ;AFTER THE ADDITION
```

AAD
ASCII Adjust for Division

Purpose: AAD does an adjustment of the dividend in AL before a following instruction divides two unpacked decimal operands, so that the result of the division is an unpacked decimal quotient.

Format: AAD (no operands)

Remarks: The high-order byte (AH) of the accumulator is multiplied by 10 and added to the low byte (AL). The result is stored into AL. AH is zeroed out.

Logic: (AL) ← (AH)*0AH + (AL)
(AH) ← 0

Flags: Affected— PF,SF,ZF
Undefined— AF,CF,OF

Encoding: 11010101 00001010
D5 0A

Example:

```
AAD   ;BEFORE THE DIVISION
```

AAM
ASCII Adjust for Multiply

Purpose: AAM does a correction of the result in AX of multiplying two unpacked decimal operands, resulting in an unpacked decimal product.

Format: AAM (no operands)

Remarks: The contents of AH are replaced by the result of dividing AL by 10. Then the contents of AL are replaced by the remainder of that division, that is, by AL modulo 10.

Logic:

(AH) ← (AL)/0AH
(AL) ← (AL)%0AH

Flags:

Affected— PF,SF,ZF
Undefined— AF,CF,OF

Encoding:

11010100 00001010
D4 0A

Example:

```
AAM    ;AFTER THE MULTIPLICATION
```

AAS
ASCII Adjust for Subtraction

Purpose: AAS does a correction of the result in the AL register of subtracting two unpacked decimal operands, resulting in an unpacked decimal difference.

Format: AAS (no operands)

Remarks: If the lower half of AL is above 9, or if the auxiliary carry flag is set, 6 is subtracted from AL and 1 is subtracted from AH. The AF and CF flags are set. The old value of AL is replaced by a byte whose upper half-byte is all zeroes and whose lower half-byte is a number from 0 to 9 created by the above subtraction.

Logic:

```
if ((AL) & 0FH) > 9 or (AF)=1 then
   (AL) <- (AL) - 6
   (AH) <- (AH) - 1
   (AF) <- 1
   (CF) <- (AF)
   (AL) <- (AL) & 0FH
```

Flags: Affected— AF,CF
Undefined— OF,PF,SF,ZF

Encoding: 00111111
3F

Example:

```
AAS   ;AFTER THE SUBTRACTION
```

ADC
Add with Carry

Purpose: ADC does an addition of the two operands, adds 1 if the CF flag is set, and returns the result to the *destination* (leftmost operand).

Format: ADC *destination,source*

Remarks: If the carry flag was set to 1, ADC adds 1 to the sum of the two operands before storing the result into the *destination* (leftmost operand). If the carry flag is set to 0, 1 is not added.

Logic: If (CF)=1, (DEST) ← (LSRC) + (RSRC) + 1
If (CF)=0, (DEST) ← (LSRC) + (RSRC)

Flags: Affected— AF, CF, OF, PF, SF, ZF

Memory or Register Operand with Register Operand:

Encoding: 000100dw modregr/m
10 + dw modregr/m

```
If d=1, LSRC = REG, RSRC = EA,DEST = REG.
If d=0, LSRC = EA, RSRC = REG, DEST = EA.
```

Example:

```
ADC   AX,SI
ADC   DI,BX
ADC   CH,BL

ADC   DX,MEM_WORD
ADC   AX,BETA[SI]
ADC   CX,ALPHA[BX][SI]

ADC   BETA[DI],BX
ADC   ALPHA[BX][SI],DI
ADC   MEM_WORD,AX
```

ADC
Add with Carry

Immediate Operand to Accumulator:

Encoding: 0001010w data
14 + w data

```
If w=0, LSRC = AL, RSRC = data, DEST = AL.
If w=1, LSRC = AX, RSRC = data, DEST = AX
```

Example:

```
ADC  AL,3
ADC  AL,VALUE_13_IMM
ADC  AX,333
ADC  AX,IMM_VAL_777
```

Immediate Operand to Memory or Register Operand:

Remarks: If an immediate-data-byte is being added to a register or memory, that byte is sign-extended to 16 bits before the addition. For this situation, the instruction byte is 83H (the s and w bits are both set).

Encoding: 100000sw mod010r/m data
80 + sw mod010r/m data

LSRC = EA,RSRC = data,DEST = EA

Example:

```
ADC   BETA[SI],4
ADC   ALPHA[BX][DI],IMM4
ADC   MEM_LOC,7396

ADC   BX,IMM_VAL_987
ADC   DH,65
ADC   CX,432
```

ADD
Addition

Purpose: ADD does an addition of the two operands and returns the result to the *destination* operand.

Format: ADD *destination,source*

Remarks: The sum of the two operands is stored into the *destination* (leftmost) operand.

Logic: (DEST) ← (LSRC) + (RSRC)

Flags: Affected— AF,CF,OF,PF,SF,ZF

Memory or Register Operand with Register Operand:

Encoding: 000000dw modregr/m
00 + dw modregr/m

```
If d=0, LSRC = REG,RSRC = EA, DEST = REG.
If d=1, LSRC = EA, RSRC = REG, DEST = EA.
```

Example: **Register to register:**

```
ADD   AX,BX
ADD   CX,DX
ADD   DI,SI
ADD   BX,BP
```

Memory to register:

```
ADD   CX,MEM_WORD
ADD   AX,BETA[SI]
ADD   DX,ALPHA[BX][DI]
```

ADD
Addition

Register to memory:

```
ADD    GAMMA[BP][DI],BX
ADD    BETA[DI],AX
ADD    MEM_WORD,CX
ADD    MEM_BYTE,BH
```

Immediate Operand to Accumulator:

Encoding: 0000010w data
04 + w data

If w=0, LSRC = AL,RSRC = data, DEST = AL.
If w=1, LSRC = AX, RSRC = data, DEST = AX.

Example:

```
ADD    AL,3
ADD    AX,456
ADD    AL,IMM_VAL_12
ADD    AX,IMM_VAL_8529
ADD    AX,IMM_VAL_6AB9H    ;DESTINATION AX
```

ADD
Addition

Immediate Operand to Memory or Register Operand:

Remarks: If an immediate-data-byte is being added from a register-or-memory word, that byte is sign-extended to 16 bits before the addition. For this situation, the instruction byte is 83H (the s and w bits are both set).

Encoding: 100000sw mod000r/m data
80 + sw mod000r/m data

LSRC = EA, RSRC = data, DEST = EA

Example: **Immediate to memory:**

```
ADD   MEM_WORD,48
ADD   GAMMA[DI],IMM_84
ADD   DELTA[BX],IMM_SENSOR_5
```

Immediate to register:

```
ADD   BX,ORIG_VAL
ADD   CX,STANDARD_COUNT
ADD   DX,1776
```

AND
Logical AND

Purpose: AND does the bit logical conjunction of the two source operands and returns the result to the *destination* operand.

Format: AND *destination,source*

Remarks: The two operands are ANDed, the result having a 1 only in those bit positions where both operands had a 1, with zeroes in all other bit positions. The result is stored into the *destination* (leftmost) operand. The carry and overflow flags are reset to 0.

Logic:
(DEST) ← (LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

Flags:
Affected— CF,OF,PF,SF,ZF.
Undefined: AF

Memory or Register Operand with Register Operand:

Encoding:
001000dw modregr/m
20 + dw modregr/m

If d=1, LSRC = REG,RSRC = EA, DEST = REG.
If d=0, LSRC = EA, RSRC = REG, DEST = EA.

AND
Logical AND

Example: **Register to register:**

```
AND    AX,BX
AND    CX,DI
AND    BH,CL
```

Memory to register:

```
AND    SI,MEM_NAME_WORD
AND    DX,BETA[BX]
AND    BX,GAMMA[BX][SI]
AND    AX,ALPHA[DI]
AND    DH,MEM_BYTE
```

Register to memory:

```
AND    MEM_NAME_WORD,BP
AND    ALPHA[DI],AX
AND    GAMMA[BX][DI],SI
AND    MEM_BYTE,AL
```

Immediate Operand to Accumulator:

Encoding: 0010010w data
24 + w data

If w=0, LSRC = AL, RSRC = data, DEST = AL.
If w=1, LSRC = AX, RSRC = data, DEST = AX.

Example:

```
AND    AL,7AH
AND    AH,0EH
AND    AX,IMM_VAL_MASK3
```

AND
Logical AND

Immediate Operand to Memory or Register Operand:

Encoding: 1000000w mod100r/m data
80 + w mod100r/m data

LSRC = EA, RSRC = data,DEST = EA

Example: **Immediate to register:**

```
AND   BL,10011110B
AND   CH,3EH
AND   DX,7A46H
AND   SI,987
```

Immediate to memory:

```
AND   MEM_WORD,7A46H
AND   MEM_BYTE,46H
AND   GAMMA[DI],IMM_MASK14
AND   CHI_BYTE[BX][SI],11100111B
```

Another example:

```
FLAGS   DB   ?
BITMASK EQU  20H
        o
        o
        o
        AND  FLAGS,0FFH-BITMASK ;TURN OFF
                                ; FLAG BIT
```

BOUND (80286)

Check Array Index Against Bounds

Purpose: Bound is used to ensure that a signed array index is within the limits defined by a 2-word block of memory.

Format: BOUND *destination,source*

Remarks: The first operand (a register) must be above or equal to the first word in the memory location referenced by the second operand, and below or equal to the second word in that memory location. If not, an INT 5 exception occurs.

Logic:

```
if ((LSRC) < (RSRC) or (LSRC) > ((RSRC)+2)
then
   (SP) ← (SP)-2
   ((SP)+1:(SP)) ← FLAGS
   (IF) ← 0
   (TF) ← 0
   (SP) ← (SP)-2
   ((SP)+1:(SP)) ← (CS)
   (CS) ← (16H)
   (SP) ← (SP)-2
   ((SP)+1:(SP)) ← (IP)
   (IP) ← (14H)
```

Flags: Affected— None
Undefined— None

BOUND (80286)
Check Array Index Against Bounds

Encoding: 01100010 modregr/m
62 modregr/m

Example:

```
            DW    ARRAY      ;ARRAY
            DW    ARRAY_END  ; DESCRIPTORS
ARRAY       DW    20 DUP(0)
ARRAY_END   EQU   $

            BOUND SI,DWORD PTR ARRAY-4
```

CALL
Call a Procedure

Purpose: CALL pushes the offset address of the next instruction onto the stack (for an inter-segment call the CS segment register is pushed first) and then transfers control to the target operand.

Direct calls and jumps can only be made to labels, relative to CS; not variables. NEAR is assumed unless FAR is stated in the instruction or in the declaration of the *target* label.

As shown in the indirect-call examples that follow, calls through variables can use the PTR operator to show the intended use of one word for NEAR calls, or two words for calls to FAR labels or procedures. Indirect calls using word registers (without square brackets) are of necessity NEAR calls.

Format: CALL *target*

Remarks: If this is an inter-segment call, the stack pointer is decremented by 2 and the contents of the CS register are pushed onto the stack. CS is then filled by second word (segment) of the doubleword inter-segment pointer.

Then the stack pointer is decremented by 2 and the contents of the Instruction Pointer are pushed onto the stack. The last step is to replace the contents of the IP by the offset of the *target* destination (the offset of the procedure's first instruction). An intra-segment or intra-group call does only steps 2, 3, and 4.

CALL
Call a Procedure

Logic:

```
1) if inter-segment then
       (SP) ← (SP) - 2
       ((SP)+1:(SP)) ← (CS)
       (CS) ← SEG
2) (SP) ← (SP) - 2
3) ((SP)+1:(SP)) ← (IP)
4) (IP) ← DEST
```

Flags: None

Direct Intra-Segment or Intra-Group:

Encoding: 11101000 disp-low disp-high
E8 disp-low disp-high

DEST = (IP) + disp

Example:

```
CALL   NEAR_LABEL
CALL   NEAR_PROC
```

Inter-Segment Direct:

Encoding: 10011010 offset-low offset-high seg-low seg-high
9A offset-low offset-high seg-low seg-high

DEST = offset, SEG = seg

Example:

```
CALL   FAR_LABEL
CALL   FAR_PROC
```

CALL
Call a Procedure

Inter-Segment Indirect:

Encoding: 11111111 mod011r/m
FF mod011r/m

DEST = (EA),SEG = (EA + 2)

Example:

```
CALL  DWORD  PTR  [BX]
CALL  DWORD  PTR  VARIABLE_NAME[SI]
CALL  MEM_DOUBLE_WORD
```

Indirect Intra-Segment or Intra-Group:

Encoding: 11111111 mod010r/m
FF mod010r/m

DEST = (EA)

Example:

```
CALL  WORD  PTR  [BX] ;BX HAS OFFSET IN DATA
                      ; SEGMENT OF WORD THAT HAS
                      ; OFFSET OF ENTRY POINT
CALL  WORD  PTR  VARIABLE_NAME
CALL  WORD  PTR  [BX][SI]
CALL  WORD  PTR  [DI]
CALL  WORD  PTR  VARIABLE_NAME[BP][SI]
CALL  MEM_WORD
CALL  BX    ;REG HAS OFFSET IN CODE
            ; SEGMENT TO ENTRY POINT
CALL  CX
```

CALL
Call a Procedure

The implicit segment register used in a register-indirect call is DS, unless BP is used or an override is specified. The implicit segment register is used to build the address that contains the offset (and segment, if a FAR call) of the call's *target*. If BP is used, the segment register SS is used. However, if a segment prefix byte is explicitly specified; such as:

```
CALL  WORD  PTR  ES:[BP][DI]
```

Then the segment register so specified is used (ES). An implicit segment register for indirect calls through variables or address-expressions is determined by the address-expression in the *source* line and the applicable ASSUME pseudo-op (see Chapter 2).

When CALL is used to transfer control, a RETurn is implied. With indirect CALLS, you must carefully ensure that the type of the CALL matches the type of RETurn, or errors can result that are difficult to trace. The issue is whether CS is saved and restored.

The type of RETurn is determined by the PROC pseudo-op.

CBW
Convert Byte to Word

Purpose: CBW does a sign extension of the AL register into the AH register.

Format: CBW (no operands)

Remarks: If the lower byte of the accumulator (AL) is less than 80H, AH is made zero. Otherwise, AH is set to FFH. This is equivalent to replicating bit 7 of AL all through AH.

Logic:

```
if (AL) < 80H then
  (AH) ← 0
else
  (AH) ← FFH
```

Flags: None

Encoding: 10011000
98

Example:

```
CBW
```

CLC
Clear Carry Flag

Purpose: CLC clears the CF flag.

Format: CLC (no operands)

Remarks: The carry flag is reset to zero.

Note: See STC for the opposite function.

Logic: (CF) ← 0

Flags: Affected— CF

Encoding: 11111000
F8

Example:

```
CLC
```

CLD
Clear Direction Flag

Purpose: CLD clears the DF flag, causing the string operations to auto-increment the operand pointers.

Format: CLD (no operands)

Remarks: The direction flag is reset to zero.

Note: See STD for the opposite function.

Logic: (DF) ← 0

Flags: Affected— DF

Encoding: 11111100
FC

Example:

```
CLD
```

CLI
Clear Interrupt Flag (Disable)

Purpose: CLI clears the IF flag and disables maskable external interrupts, which appear on the INTR line of the processor. (Non-maskable interrupts), that appear on the NMI line are not disabled.)

Format: CLI (no operands)

Remarks: The interrupt flag is reset to zero.

Note: See STI for the opposite function.

Logic: (IF) ← 0

Flags: Affected— IF

Encoding: 11111010
FA

Example:

```
CLI
```

CMC
Complement Carry Flag

Purpose: CMC complements the CF flag.

Format: CMC (no operands)

Remarks: If the carry flag is 0, it is set to 1. If it is a 1, it is reset to 0.

Logic:

```
if (CF)=0 then
   (CF) <- 1
else (CF) <- 0
```

Flags: Affected— CF

Encoding: 11110101
F5

Example:

```
CMC
```

CMP

Compare Two Operands

Purpose: CMP does a subtraction of the two operands causing the flags to be affected, but does not return the result.

Format: CMP *destination,source*

Remarks: The *source* (rightmost) operand is subtracted from the *destination* (leftmost) operand. The flags are altered but the operands remain unaffected.

The *source* (rightmost) operand must be of the same type (byte or word) as the *destination* operand. The only exception for CMP is comparing an immediate data byte with a memory word.

Logic: (LSRC) - (RSRC)

Flags: Affected— AF, CF, OF, PF, SF, ZF

Memory or Register Operand with Register Operand:

Encoding: 001110dw modregr/m
38 + dw modregr/m

If d=1, LSRC = REG, RSRC = EA.
If d=0, LSRC = EA, RSRC = REG.

CMP
Compare Two Operands

Example: **Register with register:**

```
CMP   AX,DX
CMP   SI,BP
CMP   BH,CL
```

Register with memory:

```
CMP   MEM_WORD,SI
CMP   MEM_BYTE,CH
CMP   ALPHA[DI],DX
CMP   BETA[BX][SI],CX
```

Memory with register:

```
CMP   DI,MEM_WORD
CMP   CH,MEM_WORD
CMP   AX,GAMMA[BP][SI]
```

Immediate Operand with Accumulator:

Encoding: 0011110w data
3C + w data

If w=0, LSRC = AL, RSRC = data.
If w=1, LSRC = AX, RSRC = data.

Example:

```
CMP   AL,6
CMP   AL,IMM_VALUE_DRIVE
CMP   AX,IMM_VAL_909
CMP   AX,999
```

CMP
Compare Two Operands

Immediate Operand with Memory or Register Operand:

Remarks: If an immediate-data-byte is being compared from a register-or-memory word, that byte is sign-extended to 16 bits before the compare. For this situation, the instruction byte is 83H (the s and w bits are both set).

Encoding:

100000sw mod111r/m data
80 + sw mod111r/m data

LSRC = EA, RSRC = data

Example: Immediate with register:

```
CMP    BH,7
CMP    CL,19_IMM_BYTE
CMP    DX,IMM_DATA_WORD
CMP    SI,798
```

Immediate with memory:

```
CMP    MEM_WORD,IMM_DATA_BYTE
CMP    GAMMA[BX],IMM_BYTE
CMP    [BX][DI],6ACEH
```

CMPS/CMPSB/CMPSW
Compare Byte or Word String

Purpose: CMPS subtracts the byte (or word) operand addressed by DI from the operand addressed by SI and affects the flags but does not return the result. As a repeated operation this provides for comparing two strings. With the appropriate repeat prefix it is possible to determine after which string element the two strings become unequal, thereby establishing an ordering between the strings.

Note that the operand indexed by DI is the rightmost operand in this instruction, and that this operand is addressed using the ES register only. You cannot override this default.

Format: CMPS *source-string,dest-string*
or
CMPSB (no operands)
or
CMPSW (no operands)

Remarks: The *dest-string* rightmost operand, using DI as an index into the extra segment is subtracted from the *source-string* (leftmost operand), which uses SI as an index. (This is the only string instruction where the DI-indexed operand appears as the rightmost operand.) Only the flags are affected, not the operands. SI and DI are then incremented, if the direction flag is reset (zero), or they are decremented, if DF=1. (See the CLD and STD instructions.) They thus point to the next element of the strings being compared. The increment is 1 for byte strings, 2 for word strings.

CMPS/CMPSB/CMPSW

Compare Byte or Word String

Logic:

```
(LSRC) - (RSRC)
if (DF) = 0 then
  (SI) <- (SI) + DELTA
  (DI) <- (DI) + DELTA
else
  (SI) <- (SI) - DELTA
  (DI) <- (DI) - DELTA
```

Flags: Affected— AF,CF,OF,PF,SF,ZF

Encoding: 1010011w
A6 + w

If w=0, LSRC = (SI), RSRC = (DI),
DELTA = 1 (BYTE).
If w=1, LSRC = (SI)+1:(SI), RSRC = (DI)+1:(DI),
DELTA = 2 (WORD).

Example:

```
MOV  SI,OFFSET STRING1
MOV  DI,OFFSET STRING2
CMPS STRING1,STRING2
    or
CMPS DS:BYTE PTR[SI],ES:[DI]
    or
CMPSB
```

The string instructions are unusual in several aspects:

1. Load SI with the offset of the source string.
2. Load DI with the offset of the destination string.

CMPS/CMPSB/CMPSW
Compare Byte or Word String

3. Each can be coded with or without symbolic memory operands.

   - If symbolic operands are coded, the assembler can check the addressability of them for you.

   - References that use hardware defaults should be coded using the operand-less forms (CMPSB and CMPSW), to avoid the additional pointer information.

   - Never use [BX] or [BP] addressing modes with the string instructions.

   - If operands are coded, only the *source* segment, DS, is overridable. The *destination* segment, ES, is not overridable.

4. If the instruction mnemonic is coded without operands, the segment registers are as follows:

   - SI defaults to an offset in the segment addressed by DS.

   - DI is required to be an offset in the segment addressed by ES.

CWD
Convert Word to Doubleword

Purpose: CWD does a sign extension of the AX register into the DX register. See the DIV instruction in this chapter.

Format: CWD (no operands)

Remarks: The high-order bit of AX is replicated throughout DX.

Logic:

```
if (AX) < 8000H then
   (DX) ← 0
else (DX) ← FFFFH
```

Flag: None

Encoding: 10011001
99

Example:

```
CWD
```

DAA

Decimal Adjust for Addition

Purpose: DAA does a correction of the result in AL of adding two packed decimal operands, resulting in a packed decimal sum.

Format: DAA (no operands)

Remarks: If the lower half-byte (4 bits) of AL is greater than 9 or if the auxiliary carry flag has been set, 6 is added to AL and AF is set. If AL is greater than 9FH or if the carry flag has been set, 60H is added to AL and CF is set to 1.

Logic:

```
if $((AL) \& 0FH) > 9$ or $(AF) = 1$ then
  $(AL) \leftarrow (AL) + 6$
  $(AF) \leftarrow 1$
if $(AF) > 9FH$ or $(CF) = 1$ then
  $(AL) \leftarrow (AL) + 60H$
  $(CF) \leftarrow 1$
```

Flags: Affected— AF,CF,PF,SF,ZF
Undefined— OF

Encoding: 00100111
27

Example:

```
DAA
```

DAS
Decimal Adjust for Subtraction

Purpose: DAS does a correction of the result in the AL register of subtracting two packed decimal operands, resulting in a packed decimal difference.

Format: DAS (no operands)

Remarks: If the lower half-byte (4 bits) of AL is greater than 9 or if the auxiliary flag has been set, 60H is subtracted from AL and CF is set.

Logic:

```
if ((AL) & 0FH) > 9 or (AF) = 1 then
  (AL) <- (AL) - 6
  (AF) <- 1
if (AL) > 9FH or (CF) = 1 then
  (AL) <- (AL) - 60H
  (CF) <- 1
```

Flags: Affected— AF, CF, PF, SF, ZF

Encoding: 00101111
2F

Example:

```
DAS
```

DEC
Decrement Destination by One

Purpose: DEC does a subtraction of 1 from the operand and returns the result to that operand.

Format: DEC *destination*

Remarks: The specified operand, *destination*, is reduced by 1.

Logic: (DEST) <- (DEST) - 1

Flags: Affected— AF, OF, PF, SF, ZF

Register Operand (Word):

Encoding: 01001 reg
48 + reg

DEST=REG

Example:

```
DEC AX
DEC DI
DEC SI
```

DEC
Decrement Destination by One

Memory or Register Operand:

Encoding: 1111111w mod001r/m
FE + w mod001r/m

DEST=EA

Example:

```
DEC   MEM_BYTE
DEC   MEM_BYTE[DI]
DEC   MEM_WORD
DEC   ALPHA[BX][SI]
DEC   BL
DEC   CH
```

DIV
Division, Unsigned

Purpose: DIV does an unsigned division of the double-length NUMR operand, contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified *source* operand. It returns the single-length quotient (QUO operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DX for 16-bit operation).

Format: DIV *source*

Remarks: If the quotient is greater than MAX (when division by zero is attempted), QUO and REM are undefined, and a type 0 interrupt is generated. Flags are undefined in any DIV operation. Non-integral quotients are truncated to integers.

If the division results in a value larger than can be held by appropriate registers, an interrupt of type 0 is generated. Flags are pushed onto the stack, IF and TF are reset to 0, and the CS register contents are pushed onto the stack. CS is then filled by the word at location 2. The current IP is pushed onto the stack and IP is then filled with the word at 0. This sequence thus includes a FAR call to the interrupt handling procedure whose segment and offset are stored at locations 2 and 0.

If the division result can fit in the appropriate registers, the quotient is stored in AL or AX (for word operands) and the remainder in AH or DX, respectively.

DIV
Division, Unsigned

Logic:

```
(temp) <- (NUMR)
if (temp)/(DIVR) > MAX,
  THE FOLLOWING SEQUENCE:
    (QUO),(REM) undefined
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- FLAGS
    (IF) <- 0
    (TF) <- 0
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- (CS)
    (CS) <- (2)
;CONTENTS OF LOCATIONS 2 AND 3
    (SP) <- (SP) - 2
    ((SP)+1:(SP)) <- (IP)
    (IP) <-  (0)
;CONTENTS OF LOCATIONS 0 AND 1
else
    (QUO) <- (temp)/(DIVR)
    (REM) <- (temp)%(DIVR)
```

Flags: Affected— No valid flags result
Undefined— AF, CF, OF, PF, SF, ZF

Encoding: 1111011w mod110r/m
F6 + w mod110r/m

If w=0, NUMR=AX, DIVR=EA, QUO=AL,
REM=AH, MAX=FFH.
If w=1, NUMR=DX:AX, DIVR=EA, QUO=AX,
REM=DX, MAX=FFFH.

In the following examples, each memory operand can be any variable or valid expression so long as its type is the same. For example, the NUMERATOR__WORD in the first example could be replaced by the expression:

```
ARRAY_NAME[BX][SI] + 67
```

when ARRAY_NAME is of type WORD. Similarly DIVISOR__BYTE could be:

```
RATE_TABLE[BP][DI]
```

when RATE_TABLE is of type BYTE.

Example: To divide a word by a byte:

```
MOV   AX,NUMERATOR_WORD
DIV   DIVISOR_BYTE
;QUOTIENT IN AL, REMAINDER IN AH
```

To divide a byte by a byte:

```
MOV   AL,NUMERATOR_BYTE
CBW     ;CONVERTS BYTE IN AL TO WORD IN AX
DIV   DIVISOR_BYTE
;QUOTIENT IN AL, REMAINDER IN AH
```

To divide a doubleword by a word:

```
MOV   DX,NUMERATOR_HI_WORD
MOV   AX,NUMERATOR_LO_WORD
DIV   DIVISOR_WORD
;QUOTIENT IN AX REMAINDER IN DX
```

To divide a word by a word:

```
MOV   AX,NUMERATOR_WORD
CWD     ;CONVERTS WORD TO DOUBLEWORD
DIV   DIVISOR_WORD
;QUOTIENT IN AX, REMAINDER IN DX
```

ENTER (80286)
Make Stack Frame for Procedure Parameters

Purpose: ENTER is used to create the stack frame required by most block-structured high-level languages.

Format: ENTER *immediate-word,immediate-byte*

Remarks: The first operand specifies how many bytes of dynamic storage are to be allocated on the stack for the routine being entered. The second operand gives the nesting level of the routine within the high level language source code. ENTER determines how many stack frame pointers are copied into the new stack frame from the preceding frame. BP is used as the current stack frame pointer.

If the second operand is 0, ENTER pushes BP, sets BP to SP, and subtracts the first operand from SP.

Logic:

```
(SP) ← (SP)-2
((SP)+1:(SP)) ← (BP)
(FP) ← (SP)
if LEVEL > 0 then
  Repeat (Level-1) times
    (BP) ← (BP)-2
    (SP) ← (SP)-2
    ((SP)+1:(SP)) ← (BP)
  End Repeat
  (SP) ← (SP)-2
  ((SP)+1:(SP)) ← (FP)
End if
(BP) ← (FP)
(SP) ← (SP) - (LSRC)
```

ENTER (80286)
Make Stack Frame for Procedure Parameters

Flags: Affected— None
Undefined— None

Encoding: 11001000 data-low data-high
C8 data-low data-high

Example: A procedure with 12 bytes of local variables would have an

```
ENTER    12,0
```

at its entry point and a LEAVE instruction before every RET. The 12 local bytes world be addressed as negative offsets from [BP].

ESC
Escape

Purpose: The ESC instruction provides a mechanism by which other processors can receive their instructions from the instruction stream and use the addressing modes. The processor does no operation for the ESC instruction other than to access a memory operand and place it on the bus.

Format: ESC *external-opcode,source*

Logic:

if mod ≠ 11, then data bus ← (EA)
if mod = 11, no operation.

Flags: None

Encoding:

11011xxx modxxxr/m
D8 + xxx modxxxr/m

Example:

```
ESC EXTERNAL_OPCODE, ADDRESS
```

Note: This opcode is a 6-bit number, which is split into the two 3-bit fields shown as xxx above.

F2XM1 (8087)
2 to the X power -1

Purpose: F2XM1 calculates the function $Y = 2^X - 1$. X is taken from the top of the floating-point stack and must be in the range $0 \leq X \leq 0.5$. The result Y replaces X at the top of the floating-point stack.

Format: F2XM1 (no operands)

Remarks: F2XM1 produces an accurate result even when X is close to zero. To obtain $Y = 2^X$, add 1 to the result. Values other than 2 can be raised to a power of X. For example:

$$10^X = 2^{X * \log_2 10}$$
$$e^X = 2^{X * \log_2 e}$$
$$Y^X = 2^{X * \log_2 Y}$$

The 8087 has instructions, described in this chapter, for loading the constants LOG_2 10 and LOG_2 e, and the FYL2X can be used to calculate $X * \log_2 Y$.

> **Note:** See FYL2X, FLDL2T, and FLDL2E for related information.

Logic: $ST \leftarrow 2^{ST} - 1$

Exception Flags: U,P (operands not checked)

Encoding: 10011011 11011001 11110000
9B D9 F0

Example:

```
F2XM1          ;Y = 2 TO THE X POWER MINUS 1
```

FABS (8087)
Absolute value

Purpose: FABS changes the 8087 top stack element to its absolute value.

Format: FABS (no operands)

Remarks: FABS makes the sign of the 8087 top stack element positive.

Logic: None

Exception Flags: I

Encoding:

```
10011011 11011001 11100001
   9B       D9       E1
```

Example:

```
FABS
```

FADD (8087)
Add Real

Purpose: FADD adds the *source* and *destination* operands and returns the sum to the *destination*.

Format: FADD (no operands)
or
FADD *source*
or
FADD *destination,source*

Remarks: The sum of the two operands is stored in the *destination* (leftmost) operand. FADD can be written without operands, with only a *source* or with a *destination* and a *source*.

> **Note:** See FADDP and FIADD for related information.

Exception Flags: I,D,O,U,P

FADD (no operands) Stack form:

Format: FADD [ST(1),ST]

> **Note:** ST(1),ST are implied operands; they are not coded, and are shown here for information only.

Remarks: FADD picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

> **Note:** FADD (no operands) is similar to the FADDP instruction with ST(*i*) being ST(1).

FADD (8087)
Add Real

Logic: ST(1) ← ST(1) + ST
pop 8087 stack

Encoding: 10011011 11011110 11000001
9B DE C1

Example:

```
FADD                    ;ADD REAL
```

FADD (source) Real memory form:

Format: FADD short__real
or
FADD long__real

Remarks: The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this form of the instruction.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← ST + mem-op

Encoding: SHORT__REAL
10011011 11011000 mod000r/m disp-low disp-high
9B D8 mod000r/m disp-low disp-high

Example:

```
FADD   SHORT_REAL
```

Encoding: LONG__REAL
10011011 11011100 mod000r/m disp-low disp-high
9B DC mod000r/m disp-low disp-high

FADD (8087)
Add Real

Example:

```
FADD    LONG_REAL
```

FADD (destination,source) Register form:

Remarks: Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format: FADD ST,ST(*i*)

Logic: ST ← ST + ST(*i*)

Encoding: 10011011 11011000 11000(*i*)
9B D8 C0 + (*i*)

Example:

```
FADD    ST,ST(1)        ;ADD REAL
FADD    ST,ST(0)        ;DOUBLE TOP OF STACK
FADD    ST,ST(7)
```

Format: FADD ST(*i*),ST

Logic: ST(*i*) ← ST(*i*) + ST

Encoding: 10011011 11011100 11000(*i*)
9B DC C0 + (*i*)

Example:

```
FADD    ST(1),ST
FADD    ST(7),ST
```

FADDP (8087)
Add Real and Pop

Purpose: FADDP adds the *source* and *destination* operands and returns the sum to the *destination*, and pops the 8087 stack.

Format: FADDP *destination,source*

Remarks: The sum of the two operands is stored into the *destination* (leftmost) operand. FADDP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. FADDP can be used to get the 8087 stack top as the *source* operand,and then discard it by popping the 8087 stack.

> **Note:** FADD (no operands) is similar to the FADDP instruction with ST(*i*) being ST(1).

Logic: ST(*i*) ← ST(*i*) + ST
pop 8087 stack

Exception Flags: I,D,O,U,P

Encoding: 10011011 11011110 11000(*i*)
9B DE C0 + (*i*)

Example:

```
FADDP  ST(7),ST      ;ADD REAL AND POP
```

FBLD (8087)
Packed Decimal (BCD) Load

Purpose: FBLD does a packed decimal to temporary real conversion and loads (pushes) the result onto the 8087 stack.

Format: FBLD *source*

Remarks: FBLD converts the content of the *source* operand from packed decimal to temporary real and loads (pushes) the result onto the 8087 stack. The sign of the *source* is preserved, including negative zero. FBLD is an exact operation; the *source* is loaded with no rounding error.

The packed decimal digits of the *source* are assumed to be in the range (0-9)H. The instruction does not check for invalid digits (A-F)H and the result of trying to load an invalid digit is undefined.

Logic: push 8087 stack
ST ← mem-op

Exception Flags: I

Encoding:
```
10011011 11011111 mod100r/m disp-low disp-high
   9B       DF    mod100r/m disp-low disp-high
```

Example:

```
FBLD   PACKED_DECIMAL ;PACKED DECIMAL (BCD) LOAD
```

FBSTP (8087)
Packed Decimal (BCD) Store and Pop

Purpose: FBSTP converts the content of the 8087 stack top to a packed decimal integer, stores the result at the *destination* in memory, and pops the 8087 stack.

Format: FBSTP *destination*

Remarks: FBSTP produces a rounded integer from a non-integral value by adding 0.5 to the value and then setting the fractional part to zero, resulting in the nearest integer.

> **Note:** Users who are concerned about rounding can precede FBSTP with FRNDINT.

Logic: mem-op ← ST
pop 8087 stack

Exception Flags: I

Encoding:

```
10011011 11011111 mod110r/m disp-low disp-high
   9B       DF    mod110r/m disp-low disp-high
```

Example:

```
FBSTP  PACKED_DECIMAL
             ;PACKED DECIMAL (BCD) STORE AND POP
```

FCHS (8087)
Change Sign

Purpose: FCHS reverses the sign of the 8087 top stack element.

Format: FCHS (no operands)

Remarks: FCHS complements the sign of the 8087 top stack element.

Logic: ST ← -ST

Exception Flags: I

Encoding: 10011011 11011001 11100000
9B D9 E0

Example:

```
FCHS                    ;CHANGE SIGN
```

FCLEX (8087)
Clear Exceptions

Purpose: FCLEX clears all exception flags, the interrupt request flag and the busy flag in the status word.

Format: FCLEX (no operands)

Remarks: The 8087's INT and BUSY lines go inactive.

FNCLEX is the alternate "no-wait" form for the FCLEX instruction.

> Note: An exception handler must issue this instruction before returning to the interrupted computation, or another interrupt request is generated immediately, and an endless loop can result.

Logic: Clear 8087 exceptions

Exception Flags: None

Encoding: 10011011 11011011 11100010
9B DB E2

Example:

```
FCLEX                   ;CLEAR EXCEPTIONS
```

FCOM (8087)
Compare Real

Purpose: FCOM compares the 8087 stack top to the *source* operand.

Format: FCOM (no operands)
or
FCOM *source*

Remarks: The *source* operand can be a register on the 8087 stack, or a short or long real memory operand. If an operand is not coded, ST is compared to ST(1).

> **Note:** Positive and negative forms of zero compare identically as if they were unsigned.

Logic: Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST>*source* then C3=0 and C0=0
If ST<*source* then C3=0 and C0=1
If ST=*source* then C3=1 and C0=0
If ST?*source* then C3=1 and C0=1

> **Note:** NAN's and ∞ cannot be compared, return C3=1 and C0=1 as shown above.

Exception Flags: I,D

FCOM (8087)
Compare Real

8087 stack top with Memory short__real:

Format: FCOM short__real

Remarks: ST is compared to a short real memory operand.

Encoding: 10011011 11011000 mod010r/m disp-low disp-high
9B D8 mod010r/m disp-low disp-high

Example:

```
FCOM    SHORT_REAL
```

8087 stack top with Memory long__real:

Format: FCOM long__real

Remarks: ST is compared to a long real memory operand.

Encoding: 10011011 11011100 mod010r/m disp-low disp-high
9B DC mod010r/m disp-low disp-high

Example:

```
FCOM    LONG_REAL
```

FCOM (8087)
Compare Real

8087 stack top with Register on 8087 stack:

Format: FCOM (no operand)

Remarks: ST is compared to ST(1)

Encoding: 10011011 11011000 11010001
9B D8 D1

Example:

```
FCOM                    ;COMPARE REAL
```

Format: FCOM ST(*i*)

Remarks: ST is compared to ST(*i*)

Encoding: 10011011 11011000 11010(*i*)
9B D8 D0 + (*i*)

Example:

```
FCOM    ST(7)           ;COMPARE REAL
```

FCOMP (8087)
Compare Real and Pop

Purpose: FCOMP compares the 8087 stack top to the *source* operand and pops the 8087 stack.

Format: FCOMP (no operands)
or
FCOMP *source*

Remarks: FCOMP operates like FCOM, and also pops the 8087 stack. The *source* operand can be a register on the 8087 stack, or a short or long real memory operand. If an operand is not coded, ST is compared to ST(1).

Note: Positive and negative forms of zero compare identically as if they were unsigned.

Logic: Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST>*source* then C3=0 and C0=0
If ST<*source* then C3=0 and C0=1
If ST=*source* then C3=1 and C0=0
If ST?*source* then C3=1 and C0=1

Note: NANs and ∞ cannot be compared and return C3=1 and C0=1 as shown above.

Exception Flags: I,D

FCOMP (8087)

Compare Real and Pop

8087 stack top with Memory short__real:

Format: FCOMP short__real

Remarks: ST is compared to a short real memory operand.

Encoding: 10011011 11011000 mod011r/m disp-low disp-high
9B D8 mod011r/m disp-low disp-high

Example:

```
FCOMP  SHORT_REAL ;COMPARE REAL AND POP
```

8087 stack top with Memory long__real:

Format: FCOMP long__real

Remarks: ST is compared to a long real memory operand.

Encoding: 10011011 11011100 mod011r/m disp-low disp-high
9B DC mod011r/m disp-low disp-high

Example:

```
FCOMP  LONG_REAL    ;COMPARE REAL AND POP
```

FCOMP (8087)
Compare Real and Pop

8087 stack top with Register on 8087 stack:

Format: FCOMP (no operand)

Remarks: ST is compared to ST(1)

Encoding: 10011011 11011000 11011001
9B D8 D9

Example:

```
FCOMP                  ;COMPARE REAL AND POP
```

Format: FCOMP ST(*i*)

Remarks: ST is compared to ST(*i*)

Encoding: 10011011 11011000 11011(*i*)
9B D8 D8 + (*i*)

Example:

```
FCOMP ST(1)            ;COMP REAL AND POP
                       ;SAME AS FCOMP (no operands)
FCOMP ST(7)            ;COMP REAL AND POP
```

FCOMPP (8087)

Compare Real and Pop Twice

Purpose: FCOMPP compares the 8087 stack top to ST(1) and pops the 8087 stack twice.

Format: FCOMPP (no operands)

Remarks: FCOMPP operates like FCOM and also pops the 8087 stack twice, discarding both operands. ST is compared to ST(1).

Note: Positive and negative forms of zero compare identically as if they were unsigned.

Logic: Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST>ST(1) then C3=0 and C0=0
If ST<ST(1) then C3=0 and C0=1
If ST=ST(1) then C3=1 and C0=0
If ST?ST(1) then C3=1 and C0=1

Note: NANs and ∞ cannot be compared and return C3=1 and C0=1 as shown above.

Exception Flags: I,D

Encoding: 10011011 11011110 11011001
9B DE D9

Example:

```
FCOMPP                  ;COMPARE REAL AND POP TWICE
```

FDECSTP (8087)
Decrement 8087 Stack Pointer

Purpose: FDECSTP subtracts 1 from the 8087 stack top pointer (TOP) in the status word.

Format: FDECSTP (no operands)

Remarks: FDECSTP does not change tags or register contents, nor does it transfer data. FDECSTP is not equivalent to pushing the 8087 stack.

> Note: Decrementing the 8087 stack pointer when TOP=0 produces TOP=7.

Logic: TOP ← TOP - 1

Exception Flags: None

Encoding: 10011011 11011001 11110110
9B D9 F6

Example:

```
FDECSTP             ;DECREMENT 8087 STACK POINTER
```

FDISI (8087)
Disable Interrupts

Purpose: FDISI sets the interrupt enable mask in the control word and prevents the 8087 from issuing an interrupt request.

Format: FDISI (no operands)

Remarks: FDISI disables interrupts.

Notes:

1. If WAIT is decoded with pending exceptions, the 8087 generates an interrupt, masked or not.
2. FNDISI is the alternate "no-wait" form for the FDISI instruction. Refer to FNDISI, FENI, FNENI, and the 8087 control word for related information.

Logic: (IEM)=1

Exception Flags: None

Encoding:

```
10011011 11011011 11100001
   9B       DB       E1
```

Example:

```
FDISI                   ;DISABLE INTERRUPTS
```

FDIV (8087)
Divide Real

Purpose: FDIV divides the *destination* by the *source* and returns the quotient to the *destination*.

Format: FDIV (no operands)
or
FDIV *source*
or
FDIV *destination,source*

Remarks: FDIV can be written without operands, with only a *source* or with a *destination* and a *source*.

Exception Flags: I,D,Z,O,U,P

FDIV (no operands) 8087 stack form:

Format: FDIV [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded, and are shown here for information only.

Remarks: FDIV picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FDIV (no operands) is similar to the FDIVP instruction with ST(i) being ST(1).

Logic: $ST(1) \leftarrow ST(1) \div ST$
pop 8087 stack

FDIV (8087)
Divide Real

Encoding: 10011011 11011110 11111001
9B DE F9

Example:

```
FDIV                    ;DIVIDE REAL
```

FDIV (source) Real memory form:

Format: FDIV short__real
or
FDIV long__real

Remarks: The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this form of the instruction.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← ST ÷ mem-op

Encoding: SHORT__REAL
10011011 11011000 mod110r/m disp-low disp-high
9B D8 mod110r/m disp-low disp-high

Example:

```
FDIV   SHORT_REAL   ;DIVIDE REAL
```

Encoding: LONG__REAL
10011011 11011100 mod110r/m disp-low disp-high
9B DC mod110r/m disp-low disp-high

Example:

```
FDIV   LONG_REAL    ;DIVIDE REAL
```

FDIV (8087)
Divide Real

FDIV (destination,source) Register form:

Remarks: Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format: FDIV ST,ST(*i*)

Logic: ST ← ST ÷ ST(*i*)

Encoding: 10011011 11011000 11110(*i*)
9B D8 F0 + (*i*)

Example:

```
FDIV   ST,ST(1)     ;DIVIDE REAL
FDIV   ST,ST(7)     ;DIVIDE REAL
```

Format: FDIV ST(*i*),ST

Logic: ST(*i*) ← ST(*i*) ÷ ST

Encoding: 10011011 11011100 11111(*i*)
9B DC F8 + (*i*)

Example:

```
FDIV   ST(1),ST     ;DIVIDE REAL
FDIV   ST(7),ST     ;DIVIDE REAL
```

FDIVP (8087)
Divide Real and Pop

Purpose: FDIVP divides the *destination* by the *source* and returns the quotient to the *destination*, and pops the 8087 stack.

Format: FDIVP *destination,source*

Remarks: FDIVP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. FDIVP can be used to get the 8087 stack top as the *source* operand, and then discard it by popping the 8087 stack.

> **Note:** FDIV (no operands) is similar to the FDIVP instruction with ST(*i*) being ST(1).

Logic: $ST(i) \leftarrow ST(i) \div ST$
pop 8087 stack

Exception Flags: I,D,Z,O,U,P

Encoding: 10011011 11011110 11111(*i*)
9B DE F8 + (*i*)

Example:

```
FDIVP  ST(1),ST    ;DIVIDE REAL AND POP
FDIVP  ST(7),ST    ;DIVIDE REAL AND POP
```

FDIVR (8087)
Divide Real Reversed

Purpose: FDIVR is a reversed division instruction. FDIVR divides the *source* operand by the *destination* operand and returns the quotient to the *destination*.

Format: FDIVR (no operands)
or
FDIVR *source*
or
FDIVR *destination,source*

Remarks: FDIVR can be written without operands, with only a *source* or with a *destination* and a *source*.

Exception Flags: I,D,Z,O,U,P

FDIVR (no operands) 8087 stack form:

Format: FDIVR [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded, and are shown here for information only.

Remarks: FDIVR picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FDIVR (no operands) is similar to the FDIVRP instruction with ST(*i*) being ST(1).

Logic: ST(1) ← ST ÷ ST(1)
pop 8087 stack

FDIVR (8087)
Divide Real Reversed

Encoding: 10011011 11011110 11110001
9B DE F1

Example:

```
FDIVR                  ;DIVIDE REAL REVERSED
```

FDIVR source Real memory form:

Format: FDIVR short__real
or
FDIVR long__real

Remarks: The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this form of the instruction.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← mem-op ÷ ST

Encoding: SHORT__REAL
10011011 11011000 mod111r/m disp-low disp-high
9B D8 mod111r/m disp-low disp-high

Example:

```
FDIVR  SHORT_REAL  ;DIVIDE REAL REVERSED
```

Encoding: LONG__REAL
10011011 11011100 mod111r/m disp-low disp-high
9B DC mod111r/m disp-low disp-high

Example:

```
FDIVR  LONG_REAL   ;DIVIDE REAL REVERSED
```

FDIVR (8087)
Divide Real Reversed

FDIVR (destination,source) Register form:

Remarks: Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format: FDIVR ST,ST(*i*)

Logic: ST ← ST(*i*) ÷ ST

Encoding: 10011011 11011000 11111(*i*)
9B D8 F8 + (*i*)

Example:

```
FDIVR  ST,ST(1)     ;DIVIDE REAL REVERSED
FDIVR  ST,ST(7)     ;DIVIDE REAL REVERSED
```

Format: FDIVR ST(*i*),ST

Logic: ST(*i*) ← ST ÷ ST(*i*)

Encoding: 10011011 11011100 11110(*i*)
9B DC F0 + (*i*)

Example:

```
FDIVR  ST(1),ST     ;DIVIDE REAL REVERSED
FDIVR  ST(7),ST     ;DIVIDE REAL REVERSED
```

FDIVRP (8087)
Divide Real Reversed and Pop

Purpose: FDIVRP is a reversed division instruction. FDIVRP divides the *source* operand by the *destination* operand and returns the quotient to the *destination*, and pops the 8087 stack.

Format: FDIVRP *destination,source*

Remarks: FDIVRP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. FDIVRP can be used to get the 8087 stack top as the *source* operand,and then discard it by popping the 8087 stack.

> **Note:** FDIVR (no operands) is similar to the FDIVRP instruction with ST(*i*) being ST(1).

Logic: ST(*i*) ← ST ÷ ST(*i*)
pop 8087 stack

Exception Flags: I,D,Z,O,U,P

Encoding: 10011011 11011110 11110(*i*)
9B DE F0 + (*i*)

Example:

```
FDIVRP ST(1),ST  ;DIVIDE REAL REVERSED AND POP
FDIVRP ST(7),ST  ;DIVIDE REAL REVERSED AND POP
```

FENI (8087)
Enable Interrupts

Purpose: FENI clears the interrupt enable mask in the control word, allowing the processor to generate interrupt requests.

Format: FENI (no operands)

Remarks: FENI allows the 8087 to generate interrupt requests.

Note: FNENI is the alternate "no-wait" form for the FENI instruction. See FNENI, FDISI, FNDISI, and 8087 control word for related information.

Logic: (IEM)=0

Exception Flags: None

Encoding:

```
10011011 11011011 11100000
   9B       DB       E0
```

Example:

```
FENI                    ;ENABLE INTERRUPTS
```

FFREE (8087)
Free Register

Purpose: FFREE changes the *destination* register's tag to empty.

Format: FFREE *destination*

Remarks: The *destination* register's content is not affected.

> **Note:** Refer to the 8087 tag word for additional information.

Logic: TAG(i) ← 11 (empty)

Exception Flags: None

Encoding: 10011011 11011101 11000(i)
9B DD C0 + (i)

Example:

```
FFREE  ST(1)        ;FREE STACK REGISTER ONE
FFREE  ST(7)        ;FREE STACK REGISTER SEVEN
```

FIADD (8087)
Integer Add

Purpose: FIADD adds the *source* and *destination* operands and returns the sum to the *destination*.

Format: FIADD *source*

Remarks: FIADD permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction. The sum of the two operands is stored into the top of 8087 stack register ST.

> Note: Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← ST + mem-op

Exception Flags: I,D,O,P

Encoding: SHORT__INTEGER

```
10011011 11011010 mod000r/m disp-low disp-high
   9B       DA    mod000r/m disp-low disp-high
```

Example:

```
FIADD  SHORT_INTEGER ;INTEGER ADD
```

Encoding: WORD__INTEGER

```
10011011 11011110 mod000r/m disp-low disp-high
   9B       DE    mod000r/m disp-low disp-high
```

Example:

```
FIADD  WORD_INTEGER ;INTEGER ADD
```

FICOM (8087)
Integer Compare

Purpose: FICOM compares the 8087 stack top to the *source* operand.

Format: FICOM *source*

Remarks: FICOM permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. FICOM converts the *source* operand, which can reference a word or short binary integer variable, to temporary real and compares the 8087 stack top to it.

> **Note:** Positive and negative forms of zero compare identically as if they were unsigned.

Logic: Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST>*source* then C3=0 and C0=0
If ST<*source* then C3=0 and C0=1
If ST=*source* then C3=1 and C0=0
If ST?*source* then C3=1 and C0=1

> **Note:** NANs and ∞ cannot be compared and return C3=1 and C0=1 as shown above.

Exception Flags: I,D

FICOM (8087)
Integer Compare

8087 stack top with Memory short integer:

Format: FICOM short__integer

Remarks: ST is compared to a short integer memory operand.

Encoding: 10011011 11011010 mod010r/m disp-low disp-high
9B DA mod010r/m disp-low disp-high

Example:

```
FICOM  SHORT_INTEGER ;INTEGER COMPARE
```

8087 stack top with Memory word integer:

Format: FICOM word__integer

Remarks: ST is compared to a word integer memory operand.

Encoding: 10011011 11011110 mod010r/m disp-low disp-high
9B DE mod010r/m disp-low disp-high

Example:

```
FICOM  WORD_INTEGER  ;INTEGER COMPARE
```

FICOMP (8087)

Integer Compare and Pop

Purpose: FICOMP compares the 8087 stack top to the *source* operand, and pops the 8087 stack.

Format: FICOMP *source*

Remarks: FICOMP operates identically to FICOM and also discards the value in ST by popping the 8087 stack. FICOMP permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. FICOMP converts the *source* operand, which can reference a word or short binary integer variable, to temporary real and compares the 8087 stack top to it.

Note: Positive and negative forms of zero compare identically as if they were unsigned.

Logic: Following the instruction, the condition codes in the 8087 status word reflect the order of the operands as follows:

If ST>*source* then C3=0 and C0=0
If ST<*source* then C3=0 and C0=1
If ST=*source* then C3=1 and C0=0
If ST?*source* then C3=1 and C0=1

Note: NANs and ∞ cannot be compared and return C3=1 and C0=1 as shown above.

Exception Flags: I,D

FICOMP (8087)
Integer Compare and Pop

8087 stack top with Memory short integer:

Format: FICOMP short__integer

Remarks: ST is compared to a short integer memory operand.

Encoding: 10011011 11011010 mod011r/m disp-low disp-high
9B DA mod011r/m disp-low disp-high

Example:

```
FICOMP SHORT_INTEGER ;INTEGER COMPARE
```

8087 stack top with Memory word integer:

Format: FICOMP word__integer

Remarks: ST is compared to a word integer memory operand.

Encoding: 10011011 11011110 mod011r/m disp-low disp-high
9B DE mod011r/m disp-low disp-high

Example:

```
FICOMP WORD_INTEGER  ;INTEGER COMPARE
```

FIDIV (8087)
Integer Divide

Purpose: FIDIV divides the *destination* by the *source* and returns the quotient to the *destination.*

Format: FIDIV *source*

Remarks: FIDIV permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction. The quotient of the operands is stored into the top of 8087 stack register ST.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST <- ST ÷ mem-op

Exception Flags: I,D,Z,O,U,P

Encoding: SHORT_INTEGER
10011011 11011010 mod110r/m disp-low disp-high
9B DA mod110r/m disp-low disp-high

Example:

```
FIDIV  SHORT_INTEGER  ;INTEGER DIVIDE
```

Encoding: WORD_INTEGER
10011011 11011110 mod110r/m disp-low disp-high
9B DE mod110r/m disp-low disp-high

Example:

```
FIDIV  WORD_INTEGER   ;INTEGER DIVIDE
```

FIDIVR (8087)
Integer Divide Reversed

Purpose: FIDIVR, a reversed division instruction, divides the *source* operand by the *destination* operand and returns the quotient to the *destination*.

Format: FIDIVR *source*

Remarks: FIDIVR permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. Any memory addressing mode can be used to define the *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction. The quotient of the operands is stored into the top of 8087 stack register ST.

Logic: ST ← mem-op ÷ ST

Exception Flags: I,D,Z,O,U,P

Encoding: SHORT_INTEGER
10011011 11011010 mod111r/m disp-low disp-high
9B DA mod111r/m disp-low disp-high

Example:

```
FIDIVR SHORT_INTEGER  ;INTEGER DIVIDE REVERSED
```

Encoding: WORD_INTEGER
10011011 11011110 mod111r/m disp-low disp-high
9B DE mod111r/m disp-low disp-high

Example:

```
FIDIVR WORD_INTEGER   ;INTEGER DIVIDE REVERSED
```

FILD (8087)
Integer Load

Purpose: FILD converts the *source* memory operand from its binary integer format to temporary real and loads (pushes) the result onto the 8087 stack.

Format: FILD *source*

Remarks: The new 8087 stack top is tagged zero if all bits in the *source* were zero, and is tagged valid otherwise. FILD permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be a short integer, a word integer, or a long integer. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: None

Exception Flags: I

FILD (8087)
Integer Load

Encoding: SHORT_INTEGER
10011011 11011011 mod000r/m disp-low disp-high
9B DB mod000r/m disp-low disp-high

Example:

```
FILD   SHORT_INTEGER   ;INTEGER LOAD
```

Encoding: WORD_INTEGER
10011011 11011111 mod000r/m disp-low disp-high
9B DF mod000r/m disp-low disp-high

Example:

```
FILD   WORD_INTEGER    ;INTEGER LOAD
```

Encoding: LONG_INTEGER
10011011 11011111 mod101r/m disp-low disp-high
9B DF mod101r/m disp-low disp-high

Example:

```
FILD   LONG_INTEGER    ;INTEGER LOAD
```

FIMUL (8087)
Integer Multiply

Purpose: FIMUL multiplies the *destination* by the *source* and returns the product to the *destination*.

Format: FIMUL *source*

Remarks: FIMUL permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction. The product of the operands is stored into the top of 8087 stack register ST.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← ST * mem-op

Exception Flags: I,D,O,P

Encoding: SHORT_INTEGER
10011011 11011010 mod001r/m disp-low disp-high
9B DA mod001r/m disp-low disp-high

Example:

```
FIMUL  SHORT_INTEGER  ;INTEGER MULTIPLY
```

Encoding: WORD_INTEGER
10011011 11011110 mod001r/m disp-low disp-high
9B DE mod001r/m disp-low disp-high

Example:

```
FIMUL  WORD_INTEGER   ;INTEGER MULTIPLY
```

FINCSTP (8087)
Increment 8087 Stack Pointer

Purpose: FINCSTP adds 1 to the 8087 stack top pointer (TOP) in the status word.

Format: FINCSTP (no operands)

Remarks: FINCSTP does not change tags or register contents, nor does it transfer data. FINCSTP is not equivalent to popping the 8087 stack since it does not set the tag of the previous 8087 stack top to empty.

> Note: Incrementing the 8087 stack pointer when TOP=7 produces TOP=0.

Logic: TOP ← TOP + 1

Exception Flags: None

Encoding: 10011011 11011001 11110111

9B	D9	F7

Example:

```
FINCSTP                 ;INCREMENT STACK POINTER
```

FINIT (8087)
Initialize Processor

Purpose: FINIT does the functional equivalent of a hardware reset to the 8087 except that it does not affect the instruction fetch synchronization of the 8087 and the 8088.

Format: FINIT (no operands)

Remarks: FINIT sets the control word to 03FFH, empties all floating point 8087 stack elements, and clears exception flags and busy interrupts. If FINIT is executed while a previous 8087 memory referencing instruction is running, the 8087 bus cycles in progress is aborted.

> **Note:** FNINIT is the alternate "no-wait" form for the FINIT instruction. See FNINIT, 8087 control word, and 8087 initialization for additional information.

Logic: initialize 8087

Exception Flags: None

Encoding: 10011011 11011011 11100011
9B DB E3

Example:

```
FINIT                  ;INITIALIZE PROCESSOR
```

FIST (8087)
Integer Store

Purpose: FIST rounds the content of the 8087 stack top to an integer according to the RC field of the control word and transfers the result to the *destination*.

Format: FIST *destination*

Remarks: The *destination* can define a word or short integer variable.

> **Note:** Negative zero is stored in the same encoding as positive zero: 0000...00.

Logic: mem-op ← ST

Exception Flags: I,P

Encoding: SHORT__INTEGER

```
10011011 11011011 mod010r/m disp-low disp-high
  9B       DB     mod010r/m disp-low disp-high
```

Example:

```
FIST   SHORT_INTEGER   ;INTEGER STORE
```

Encoding: WORD__INTEGER

```
10011011 11011111 mod010r/m disp-low disp-high
  9B       DF     mod010r/m disp-low disp-high
```

Example:

```
FIST   WORD_INTEGER    ;INTEGER STORE
```

FISTP (8087)
Integer Store and Pop

Purpose: FISTP rounds the content of the 8087 stack top to an integer according to the RC field of the control word and transfers the result to the *destination*. FISTP pops the 8087 stack following the transfer.

Format: FISTP *destination*

Remarks: FISTP operates like FIST and also pops the 8087 stack following the transfer. The *destination* can define a word or short integer variable. The *destination* can be any of the binary integer data types.

> Note: Negative zero is stored in the same encoding as positive zero: 0000...00.

Logic:

mem-op ← ST
pop 8087 stack

Exception Flags: I,P

FISTP (8087)
Integer Store and Pop

Encoding: SHORT_INTEGER
10011011 11011011 mod011r/m disp-low disp-high
9B DB mod011r/m disp-low disp-high

Example:

```
FISTP  SHORT_INTEGER    ;INTEGER STORE AND POP
```

Encoding: WORD_INTEGER
10011011 11011111 mod011r/m disp-low disp-high
9B DF mod011r/m disp-low disp-high

Example:

```
FISTP  WORD_INTEGER     ;INTEGER STORE AND POP
```

Encoding: LONG_INTEGER
10011011 11011111 mod111r/m disp-low disp-high
9B DF mod111r/m disp-low disp-high

Example:

```
FISTP  LONG_INTEGER     ;INTEGER STORE AND POP
```

FISUB (8087)
Integer Subtract

Purpose: FISUB subtracts the *source* operand from the *destination* and returns the difference to the *destination*.

Format: FISUB *source*

Remarks: FISUB permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. Any memory addressing mode can be used to define the *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction. The difference of the two operands is stored in the top of 8087 stack register ST.

Logic: ST ← ST - mem-op

Exception Flags: I,D,O,P

Encoding: SHORT_INTEGER
10011011 11011010 mod100r/m disp-low disp-high
9B DA mod100r/m disp-low disp-high

Example:

```
FISUB  SHORT_INTEGER  ;INTEGER SUBTRACT
```

Encoding: WORD_INTEGER
10011011 11011110 mod100r/m disp-low disp-high
9B DE mod100r/m disp-low disp-high

Example:

```
FISUB  WORD_INTEGER   ;INTEGER SUBTRACT
```

FISUBR (8087)
Integer Subtract Reversed

Purpose: FISUBR is a reversed subtraction instruction. FISUBR subtracts the *destination* from the *source* and returns the difference to the *destination*.

Format: FISUBR *source*

Remarks: FISUBR permits a binary integer in memory to be used directly as a *source* operand. The *source* operand can be either a short integer or a word integer. Any memory addressing mode can be used to define the *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this instruction. The difference of the operands is stored in the top of 8087 stack register ST.

Logic: ST ← mem-op - ST

Exception Flags: I,D,O,P

Encoding: SHORT__INTEGER
10011011 11011010 mod101r/m disp-low disp-high
9B DA mod101r/m disp-low disp-high

Example:

```
FISUBR SHORT_INTEGER ;INTEGER SUBTRACT REVERSED
```

Encoding: WORD__INTEGER
10011011 11011110 mod101r/m disp-low disp-high
9B DE mod101r/m disp-low disp-high

Example:

```
FISUBR WORD_INTEGER  ;INTEGER SUBTRACT REVERSED
```

FLD (8087)
Load Real

Purpose: FLD loads (pushes) the *source* operand onto the top of the 8087 register stack.

Format: FLD *source*

Remarks: FLD loads (pushes) the *source* operand onto the top of the 8087 stack by decrementing the 8087 stack pointer by 1 and then copying the content of the *source* to the new 8087 stack top. The *source* can be a register on the 8087 stack ST(*i*) or any of the real data types in memory. Short and long real *source* operands are converted to temporary real.

> **Note:** Coding FLD ST(0) duplicates the 8087 stack top.

Exception Flags: I,D

Register operand to ST:

Logic:

```
T₁ ← ST( i)
push 8087 stack
ST ← T₁
```

Encoding: 10011011 11011001 11000(*i*) (register)
9B D9 C0 + (*i*) (register)

Example:

```
FLD ST(0) ;LOAD REAL (DUPES TOP OF STACK)
FLD ST(7) ;LOAD REAL STACK REGISTER SEVEN ST(7)
```

FLD (8087)
Load Real

Memory operand to ST:

Logic: push 8087 stack
ST ← mem-op

Encoding: SHORT_REAL
10011011 11011001 mod000r/m disp-low disp-high
9B D9 mod000r/m disp-low disp-high

Example:

```
FLD     SHORT_REAL     ;LOAD REAL
```

Encoding: LONG_REAL
10011011 11011101 mod000r/m disp-low disp-high
9B DD mod000r/m disp-low disp-high

Example:

```
FLD     LONG_REAL      ;LOAD REAL
```

Encoding: TEMP_REAL
10011011 11011011 mod101r/m disp-low disp-high
9B DB mod101r/m disp-low disp-high

Example:

```
FLD     TEMP_REAL      ;LOAD REAL
```

FLD1 (8087)
Load +1.0

Purpose: FLD1 loads (pushes) +1.0 onto the 8087 stack.

Format: FLD1 (no operands)

Remarks: FLD1 is a constant instruction that loads (pushes) a value of +1.0 onto the 8087 stack. The value has full temporary real precision of 64 bits and is accurate to approximately 19 decimal digits.

> **Note:** Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long, saving storage and improving execution speed.

Logic:

push 8087 stack
ST ← 1.0

Exception Flags: I

Encoding:

```
10011011 11011001 11101000
   9B       D9       E8
```

Example:

```
FLD1                    ;LOAD +1.0
```

FLDCW (8087)
Load Control Word

Purpose: FLDCW replaces the current processor control word with the word defined by the *source* operand.

Format: FLDCW *source*

Remarks: FLDCW is typically used to establish, or change, the 8087's mode of operation. When changing modes, the recommended procedure is to first clear any exceptions and then load the new control word.

> Note: If an exception bit in the status word is set, loading a new control word that unmasks that exception and clears the interrupt enable mask generates an immediate interrupt request before the next instruction is executed.

Logic: 8087 control word ← mem-op

Exception Flags: None

Encoding: 10011011 11011001 mod101r/m disp-low disp-high
9B D9 mod101r/m disp-low disp-high

Example:

```
FLDCW  TWO_BYTES   ;LOAD CONTROL WORD
```

FLDENV (8087)
Load Environment

Purpose: FLDENV reloads the 8087 environment from the memory area defined by the *source* operand.

Format: FLDENV *source*

Remarks: This data should have been written by a previous FSTENV/FNSTENV instruction. 8088 instructions (that do not reference the environment image) can immediately follow FLDENV, but do not execute an 8087 instruction that follows without an intervening FWAIT or assembler generated WAIT.

> **Note:** Loading an environment image that contains an unmasked exception causes an immediate interrupt request from the 8087 (assuming IEM=0 in the environment image).

Logic: 8087 environment ← mem-op

Exception Flags: None

Encoding: 10011011 11011001 mod100r/m disp-low disp-high
9B D9 mod100r/m disp-low disp-high

Example:

```
FLDENV FOURTEEN_BYTES  ;LOAD ENVIRONMENT
```

FLDL2E (8087)
Load Log$_2e$

Purpose: FLDL2E loads (pushes) the value Log$_2e$ (Log base 2 of e) onto the 8087 stack.

Format: FLDL2E (no operands)

Remarks: FLDL2E is a constant instruction that loads (pushes) a value of Log$_2e$ onto the 8087 stack. The value has full temporary real precision of 64 bits and is accurate to 19 decimal digits.

> **Note:** Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long, saving storage and improving execution speed.

Logic: push 8087 stack
ST ← Log$_2$ e

Exception Flags: I

Encoding: 10011011 11011001 11101010
9B D9 EA

Example:

```
FLDL2E                  ;LOAD LOG BASE 2 OF e
```

FLDL2T (8087)
Load $\mathrm{Log}_2 10$

Purpose: FLDL2T loads (pushes) the value $\mathrm{Log}_2 10$ (Log base 2 of 10) onto the 8087 stack.

Format: FLDL2T (no operands)

Remarks: FLDL2T is a constant instruction that loads (pushes) a value of $\mathrm{Log}_2 10$ onto the 8087 stack. The value has full temporary real precision of 64 bits and is accurate to approximately 19 decimal digits.

> **Note:** Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long, saving storage and improving execution speed.

Logic: push 8087 stack
$ST \leftarrow \mathrm{Log}_2 10$

Exception Flags: I

Encoding: 10011011 11011001 11101001
9B D9 E9

Example:

```
FLDL2T                  ;LOAD LOG BASE 2 OF 10
```

FLDLG2 (8087)
Load $\text{Log}_{10}2$

Purpose: FLDLG2 loads (pushes) the value $\text{Log}_{10}2$ (Log base 10 of 2) onto the top of the floating-point stack.

Format: FLDLG2 (NO OPERAND)

Remarks: FLDLG2 is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and executes faster while simplifying programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

> Note: Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long saving storage and improving execution speed.

Logic:

```
push 8087 stack
ST <- Log10 2
```

Exception Flags: I

Encoding:

```
10011011 11011001 11101100
   9B       D9       EC
```

Example:

```
FLDLG2                ;LOAD LOG BASE 10 OF 2
```

FLDLN2 (8087)
Load Log$_e$2

Purpose: FLDLN2 loads (pushes) the value Log$_e$2 (Log base *e* of 2) onto the top of the floating-point stack.

Format: FLDLN2 (no operands)

Remarks: FLDLN2 is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and executes faster while simplifying programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

> **Note:** Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long saving storage and improving execution speed.

Logic:

```
push 8087 stack
ST ← Log e2
```

Exception Flags: I

Encoding:

```
10011011 11011001 11101101
   9B       D9       ED
```

Example:

```
FLDLN2              ;LOAD LOG BASE e OF 2
```

FLDPI (8087)
Load PI

Purpose: FLDPI loads (pushes) the value π onto the top of the floating point stack.

Format: FLDPI (no operands)

Remarks: FLDPI is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and executes faster while simplifying programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

> **Note:** Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long saving storage and improving execution speed.

Logic: push 8087 stack
$ST \leftarrow \pi$

Exception Flags: I

Encoding: 10011011 11011001 11101011
9B D9 EB

Example:

```
FLDPI            ;LOAD PI
```

FLDZ (8087)
Load Zero

Purpose: FLDZ loads (pushes) the value +0.0 onto the top of the floating point stack.

Format: FLDZ (no operands)

Remarks: FLDZ is a constant instruction. A constant instruction loads a commonly used constant in a way that uses less memory and executes faster while simplifying programming. The constant has temporary real precision of 64 bits and accuracy of approximately 19 decimal digits.

> Note: Temporary real constants occupy 10 memory bytes, and constant instructions are only 2 bytes long saving storage and improving execution speed.

Logic: push 8087 stack
ST ← +0.0

Exception Flags: I

Encoding: 10011011 11011001 11101110
9B D9 EE

Example:

```
FLDZ                ;LOAD +0.0
```

FMUL (8087)
Multiply Real

Purpose: FMUL multiplies the *destination* operand by the *source* and returns the product to the *destination*.

Format: FMUL (no operands)
or
FMUL *source*
or
FMUL *destination,source*

Remarks: The product of the two operands is stored into the *destination* (leftmost) operand. FMUL can be written without operands, with only a *source* or with a *destination* and a source.

Exception Flags: I,D,O,U,P

FMUL (no operands) 8087 stack form:

Format: FMUL [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded, and are shown here for information only.

Remarks: FMUL picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FMUL (no operands) is similar to the FMULP instruction with ST(*i*) being ST(1).

Logic: ST(*i*) ← ST(*i*) * ST
pop 8087 stack

FMUL (8087)
Multiply Real

Encoding: 10011011 11011110 11001001
9B DE C9

Example:

```
FMUL                    ;MULTIPLY REAL
```

FMUL (source) Real memory form:

Format: FMUL short__real
or
FMUL long__real

Remarks: The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this form of the instruction.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← ST * mem-op

Encoding: SHORT__REAL
10011011 11011000 mod001r/m disp-low disp-high
9B D8 mod001r/m disp-low disp-high

Example:

```
FMUL    SHORT_REAL
```

Encoding: LONG__REAL
10011011 11011100 mod001r/m disp-low disp-high
9B DC mod001r/m disp-low disp-high

Example:

```
FMUL    LONG_REAL
```

FMUL (8087)
Multiply Real

FMUL (destination,source) Register form:

Remarks: Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format: FMUL ST,ST(*i*)

Logic: ST ← ST * ST(*i*)

Encoding: 10011011 11011000 11001(*i*)
9B D8 C8 + (*i*)

Example:

```
FMUL   ST,ST(1)        ;MULTIPLY REAL
FMUL   ST,ST(0)        ;SQUARE TOP OF STACK
FMUL   ST,ST(7)
```

Format: FMUL ST(*i*),ST

Logic: ST(*i*) ← ST(*i*) * ST

Encoding: 10011011 11011100 11001(*i*)
9B DC C8 + (*i*)

Example:

```
FMUL   ST(1),ST
FMUL   ST(7),ST
```

FMULP (8087)
Multiply Real and Pop

Purpose: FMULP multiplies the *destination* operand by the *source* and returns the product to the *destination*, and pops the floating-point stack.

Format: FMULP *destination,source*

Remarks: The product of the two operands is stored into the *destination* (leftmost) operand. FMULP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. FMULP can be used to get the 8087 stack top as the *source* operand,and then discard it by popping the 8087 stack.

> **Note:** FMUL (no operands) is similar to the FMULP instruction with ST(*i*) being ST(1).

Logic: ST(*i*) ← ST(*i*) * ST
pop 8087 stack

Exception Flags: I,D,O,U,P

Encoding: 10011011 11011110 11001(*i*)
9B DE C8 + (*i*)

Example:

```
FMULP  ST(7),ST      ;MULTIPLY REAL AND POP
```

FNCLEX (8087)
Clear Exceptions

Purpose: FNCLEX clears all exception flags, the interrupt request flag and the busy flag in the status word.

Format: FNCLEX (no operands)

Remarks: FNCLEX or FCLEX must be issued by an exception handler before it returns to the interrupted instruction. If this is not done, a new exception is immediately generated causing an endless loop to occur.

FNCLEX is the "no-wait" form of FCLEX and should only be used when there is danger of generating an infinite wait condition.

> **Note:** See FCLEX and status word for related information.

Logic: Clear 8087 exceptions

Exception Flags: None

Encoding: 11011011 11100010
DB E2

Example:

```
FNCLEX          ;CLEAR EXCEPTIONS NO WAIT
```

FNDISI (8087)
Disable Interrupts

Purpose: FNDISI sets the interrupt enable mask in the control word and prevents the processor from issuing an interrupt request.

Format: FNDISI (no operands)

Remarks: FNDISI is the "no-wait" form of FDISI and should only be used when there is danger of generating an infinite wait condition.

Notes:

1. If WAIT is decoded with pending exceptions, the 8087 generates an interrupt, masked or not.
2. See FDISI, FENI, FNENI, and the 8087 control word for related information.

Logic: (IEM)=1

Exception Flags: None

Encoding: 11011011 11100001
DB E1

Example:

```
FNDISI          ;DISABLE INTERRUPTS NO WAIT
```

FNENI (8087)
Enable Interrupts

Purpose: FNENI clears the interrupt enable mask in the control word, allowing the processor to generate interrupt requests.

Format: FNENI (no operands)

Remarks: FNENI is the "no-wait" form of this instruction. It should be used only when there is danger of generating an infinite wait condition.

> **Note:** See FENI, FDISI, FNDISI, and 8087 control word for related information.

Logic: (IEM)=0

Exception Flags: None

Encoding:
```
11011011 11100000
   DB       E0
```

Example:

```
FNENI                   ;ENABLE INTERRUPTS NO WAIT
```

FNINIT (8087)
Initialize Processor

Purpose: FNINIT does the functional equivalent of a hardware reset to the 8087 except that it does not affect the instruction fetch synchronization of the 8087 and the 8088.

Format: FNINIT (no operands)

Remarks: FNINIT sets the control word to 03FFH, empties all floating point stack elements, and clears exception flags and busy interrupts. If FNINIT is executed while a previous 8087 memory referencing instruction is running, the 8087 bus cycles in progress is aborted.

> **Note:** FNINIT is the alternate "no-wait" form for the FINIT instruction. It should only be used when there is danger of generating an infinite wait condition. See FINIT, 8087 control word, and 8087 initialization for additional information.

Logic: initialize 8087

Exception Flags: None

Encoding: 11011011 11100011
DB E3

Example:

```
FNINIT          ;INITIALIZE PROCESSOR NO WAIT
```

FNOP (8087)
No Operation

Purpose: FNOP causes no operation.

Format: FNOP (no operands)

Remarks: FNOP stores the 8087 stack top to the 8087 stack top and thus effectively does no operation. FNOP can be used to replace a deleted instruction in an assembled program. This allows the altered code to be run without reassembly.

Logic: ST ← ST

Exception Flags: None

Encoding:

```
10011011 11011001 11010000
   9B       D9       D0
```

Example:

```
FNOP                    ;NO OPERATION
```

FNSAVE (8087)
Save State

Purpose: FNSAVE writes the full 8087 state, environment plus register stack, to the memory location specified in the destination operand, and initializes the 8087.

Format: FNSAVE *destination*

Remarks: This is the "no-wait" form of FSAVE. It should only be used when there is danger of generating an infinite wait condition.

If an instruction is executing at the time a FNSAVE instruction is decoded, the instruction is allowed to complete execution before FNSAVE is performed. This means that the save always reflects the state of the processor following the completion of any active instruction.

> **Note:** See FSAVE, and 8087 save state for related information.

Logic: mem-op ← 8087 state

Exception Flags: None

Encoding: 11011101 mod110r/m disp-low disp-high
DD mod110r/m disp-low disp-high

Example:

```
FNSAVE SAVE_STATE  ;SAVE CURRENT STATE OF 8087
```

FNSTCW (8087)
Store Control Word

Purpose: FNSTCW writes the current 8087 control word to the memory location defined by destination.

Format: FNSTCW *destination*

Remarks: FNSTCW is the "no-wait" form of FSTCW. This form should only be used when there is danger of generating an infinite wait condition.

The control word stores all exception masks, the interrupt enable mask, and fields for precision control, rounding control and infinity control.

> Note: See FSTCW, and the 8087 control word for related information.

Logic: mem-op ← 8087 control word

Exception Flags: None

Encoding: 11011001 mod111r/m disp-low disp-high
D9 mod111r/m disp-low disp-high

Example:

```
FNSTCW SAVE_CONTROL     ;STORE CONTROL WORD
```

FNSTENV (8087)
Store Environment

Purpose: FNSTENV writes the 8087's basic status (control, status, and tag words) and exception pointers to the memory location defined by the *destination* operand

Format: FNSTENV *destination*

Remarks: This is the "no-wait" form of FSTENV. This form should only be used if there is danger of generating an infinite wait condition

FSTENV/FNSTENV sets all exception masks in the 8087 after the environment is saved; it does not affect the interrupt enable mask. FNSTENV must be allowed to finish before any other instruction is decoded. An explicit FWAIT should precede the next 8087 instruction. There is no risk of the following FWAIT causing an infinite wait condition because FNSTENV masks all exceptions, thus preventing an interrupt request from the 8087. If FNSTENV is decoded while another instruction is executing concurrently in the NEU (8087 numeric execution unit), the 8087 does not store the environment until the other instruction has completed. The data saved by this instruction, therefore, reflects the state of the 8087 after any previously decoded instruction has been executed.

> **Note:** See FSTENV and 8087 environment for related information.

FNSTENV (8087)
Store Environment

Logic: mem_op ← 8087 environment
set all exception masks

Exception Flags: None

Encoding: 11011001 mod110r/m disp-low disp-high
D9 mod110r/m disp-low disp-high

Example:

```
FNSTENV STORE_ENVIRONMENT  ;STORE ENVIRONMENT
```

FNSTSW (8087)
Store Status Word

Purpose: FNSTSW writes the current value of the 8087 status word to the memory location defined by the *destination* operand.

Format: FNSTSW *destination*

Remarks: This is the "no-wait" form of this instruction. This form should only be used when there is danger of generating an infinite wait condition.

Use FNSTSW or its companion instruction FSTSW to poll the status of the processor. This status information can then be used to do conditional branching. FNSTSW (no-wait) can be used to determine if the 8087 is busy.

> **Note:** See FSTSW and the 8087 status word for related information.

Logic: mem-op ← 8087 status word

Exception Flags: None

Encoding: 11011101 mod111r/m disp-low disp-high
DD mod111r/m disp-low disp-high

FNSTSW (8087)
Store Status Word

Example: This is an example showing a method to poll the status of the 8087 looking for a not busy condition.

```
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME  CS:CSEG,DS:CSEG

STATUS  DW      0       ;8087 STATUS
STAT    RECORD  STBUSY:1,STC3:1,STTOP:3,STC2:1,
STC1:1,STC0:1,STINT:1,STRES:1,STPFLG:1,STUFLG:1
,STOFLG:1,STZFLG:1,STDFLG:1,STIFLG:1
;The previous instruction
; (STAT   RECORD...STIFLG:1)
; must be contained on one line of code.

;LOOP HERE UNTIL THE 8087 IS NOT BUSY
POLL:   FNSTSW  STATUS   ;STORE STATUS WORD
        TEST    STATUS,MASK STBUSY
        JNZ     POLL

CSEG    ENDS
        END
```

FPATAN (8087)
Partial Arc Tangent

Purpose: FPATAN computes the $\theta = \text{ARCTAN}(Y/X)$ function. X is taken from the 8087 stack top element and Y from the next 8087 stack element. The instruction pops the 8087 stack and returns θ to the new 8087 stack top, overwriting the Y operand.

Format: FPATAN (no operands)

Remarks: The FPATAN instruction assumes that the operands are valid and in-range. The valid range for X and Y is:

$$0 < Y < X < +\infty$$

To be considered valid, an operand must be normalized. If an operand is either invalid or out-of-range, the instruction produces an undefined result without signalling an exception.

Logic:

```
$T_1 \leftarrow$ ARCTAN (ST(1)/ST)
pop 8087 stack
ST $\leftarrow T_1$
```

Exception Flags: U,P (operands not checked)

Encoding: 10011011 11011001 11110011

9B D9 F3

Example:

```
FPATAN          ;PARTIAL ARC TANGENT
```

FPREM (8087)
Partial Remainder

Purpose: FPREM does modulo division of the 8087 stack top element ST by the next 8087 stack element ST(1). The sign of the remainder is the same as the sign of the original dividend.

Format: FPREM (no operands)

Remarks: FPREM operates by doing successive subtractions. It can reduce a magnitude difference of up to 2^{64} in one execution. If FPREM produces a remainder that is less than the modulus ST(1), the function is complete and bit C2 of the status word condition code is cleared. If the function is incomplete, C2 is set to 1. The result in ST is then called the partial remainder.

You can use software to inspect C2 by storing the status word following execution of FPREM and re-executing the instruction (using the partial remainder in ST as the dividend), until C2 is cleared. An alternate possibility is comparing ST with ST(1) to determine when the function is complete. If ST > ST(1), FPREM must be executed again. If ST = ST(1), the remainder is 0 and execution is complete. If ST < ST(1), execution is complete and the remainder is ST. FPREM produces an exact result. The precision exception does *not* occur.

FPREM (8087)
Partial Remainder

An important use for FPREM is to reduce arguments (operands) of transcendental functions to the range permitted by these instructions. For example, the FPTAN (tangent) instruction requires its argument to be less than $\pi/4$. Using $\pi/4$ as a modulus, FPREM reduces an argument so that it is in range of FPTAN. Because FPREM produces an exact result, the argument reduction does *not* introduce round-off error into the calculation, even if several iterations are required to bring the argument into range.

FPREM also provides the least-significant three bits of the quotient generated by FPREM (in C3,C1,C0). This is also important for transcendental argument reduction because it locates the original angle in the correct one of eight $\pi/4$ segments of the unit circle.

Logic:

ST ← repeat (ST-ST(1))
If ST > ST(1) then C2 = 1, PREM = ST
If ST = ST(1) then C2 = 0, REM = 0
If ST < ST(1) then C2 = 0, REM = ST

Exception Flags: I,D,U

Encoding: 10011011 11011001 11111000
9B D9 F8

Example:

```
FPREM           ;PARTIAL REMAINDER
```

FPTAN (8087)
Partial Tangent

Purpose: FPTAN computes the Y/X = TAN (θ) function. The value θ is taken from the 8087 stack top element (ST). The result of the operation is a ratio; Y replaces θ in the 8087 stack and X is Pushed, becoming the new 8087 stack top.

Format: FPTAN (no operands)

Remarks: The FPTAN Instruction assumes that the operand is valid and in-range. The value of θ must be in the range $0 \leq \theta < \pi/4$. To be considered valid, an operand must be normalized. If the operand is invalid or out-of-range, the instruction produces an undefined result without signalling an exception.

The ratio result of FPTAN and the ratio argument of FPATAN optimize the calculation of the other trigonometric functions including SIN, COS, ARCSIN AND ARCTAN. These functions can be derived from TAN and ARCTAN using standard trigonometric identities.

Logic:

```
Y/X ← TAN(ST)
ST ← Y
push 8087 stack
ST ← X
```

Exception Flags: I,P (operands not checked)

Encoding: 10011011 11011001 11110010
9B D9 F2

Example:

```
FPTAN             ;PARTIAL TANGENT
```

FRNDINT (8087)
Round to Integer

Purpose: FRNDINT rounds the 8087 stack top element ST to an integer.

Format: FRNDINT (no operands)

Remarks: The rules for rounding are determined by the setting of the RC field of the control word. There are four modes of rounding available:

RC = 00	Round to nearest integer; if exactly midway, the even integer is chosen. (This is the default mode.)
RC = 01	Round downward (toward - ∞)
RC = 10	Round upward (toward + ∞)
RC = 11	Chop (round toward 0)

Logic: ST ← nearest integer (ST)

Exception Flags: I,P

Encoding:
```
10011011 11011001 11111100
   9B       D9       FC
```

Example:

```
FRNDINT          ;ROUND TO INTEGER
```

FRSTOR (8087)
Restore State

Purpose: FRSTOR, the restore state instruction, reloads the 8087 from the 94-byte memory area defined by *source* operand. This information should have been written by a previous FSAVE/FNSAVE instruction.

Format: FRSTOR *source*

Remarks: CPU (8088) instructions that do not reference the save image can immediately follow the FRSTOR instruction. However, any 8087 instruction following this instruction should always be preceded by an FWAIT. The saved image requires 94 bytes of memory.

> **Note:** The 8087 reacts to its new state at the finish of the FRSTOR; it generates an immediate interrupt, if the exception and mask bits in the memory image so indicate.

Logic: 8087 state ← mem-op

Exception Flags: None

Encoding: 10011011 11011101 mod100r/m disp-low disp-high
9B DD mod100r/m disp-low disp-high

Example:

```
FRSTOR SAVE_STATE        ;RESTORE STATE
```

FSAVE (8087)
Save State

Purpose: FSAVE writes the full 8087 state, environment plus register stack, to the memory location specified in the destination operand, and initializes the 8087.

Format: FSAVE *destination*

Remarks: After the image is written to memory, the 8087 is initialized as if FINIT/FNINIT had been executed.

> **Note:** See FNSAVE, FINIT, and 8087 save state for related information.

Logic: mem-op ← 8087 state

Exception Flags: None

Encoding: 10011011 11011101 mod110r/m disp-low disp-high
9B DD mod110r/m disp-low disp-high

Example:

```
FSAVE SAVE_STATE ;SAVE CURRENT STATE OF 8087
```

FSCALE (8087)
Scale

Purpose: FSCALE interprets the value contained in ST(1) as an integer, and adds this value to the exponent of the number in ST.

Format: FSCALE (no operands)

Remarks: FSCALE provides rapid multiplication or division by integral powers of 2. It is especially useful for scaling the elements of a vector.

The value contained in ST(1) must be an integer in the range $-2^{15} \leq ST(1) < 2^{15}$. If the value in ST(1) is out-of-range, or $0 < ST(1) < 1$, the instruction produces an undefined result and does *not* signal an exception. Loading the scale factor from a word integer ensures correct operation.

> **Note:** If the value is not an integer, but is in-range and is greater in magnitude than 1, FSCALE uses the nearest integer smaller in magnitude.

Logic: $ST \leftarrow ST*2^{ST(1)}$

Exception Flags: I,O,U

Encoding: 10011011 11011001 11111101
9B D9 FD

Example:

```
FSCALE              ;SCALE
```

FSQRT (8087)
Square Root

Purpose: FSQRT replaces the content of the 8087 stack top element ST with its square root.

Format: FSQRT (no operands)

Remarks: FSQRT executes rapidly and is comparable to normal division. This allows an application that would normally execute slowly because of the presence of square root calculations to achieve normal speed. ST must be in the range:
$-0 \leq ST \leq +\infty$

Note: The square root of -0 is defined to be -0.

Logic: $ST \leftarrow \sqrt{ST}$

Exception Flags: I,D,P

Encoding: 10011011 11011001 11111010
9B D9 FA

Example:

```
FSQRT              ;SQUARE ROOT
```

FST (8087)
Store Real

Purpose: FST transfers the 8087 stack top ST to the location defined by the destination operand.

Format: FST *destination*

Remarks: FST transfers the top of the 8087 stack to the destination, which can be another 8087 stack element or a short or long real memory operand. If the contents of the 8087 stack top are longer than the destination, the contents of the 8087 stack top are rounded according to the RC field in the control word. If the 8087 stack top is tagged a special value (it contains ∞, a NAN, or a denormal), the 8087 stack top significand is not rounded. In this case, the least significant bits of the 8087 stack top are deleted to fit the destination. The exponent is treated in the same way. This preserves the value's identification as a special value, so that it can be properly loaded and tagged later in the program, if desired.

Exception Flags: I,O,U,P

FST ST to Register operand:

Format: FST ST(*i*)

Remarks: Transfer ST to 8087 stack element ST(*i*).

Logic: ST(*i*) ← ST

Encoding: 10011011 11011101 11010(*i*)
9B DD D0 + (*i*)

Example:

```
FST   ST(5)      ;STORE REAL
```

FST (8087)
Store Real

FST ST to Memory operand:

Format: FST short__real
or
FST long__real

Logic: mem-op ← ST

Encoding: SHORT__REAL
10011011 11011001 mod010r/m disp-low disp-high
9B D9 mod010r/m disp-low disp-high

Example:

```
FST    SHORT_REAL            ;STORE REAL
```

Encoding: LONG__REAL
10011011 11011101 mod010r/m disp-low disp-high
9B DD mod010r/m disp-low disp-high

Example:

```
FST    LONG_REAL             ;STORE REAL
```

FSTCW (8087)
Store Control Word

Purpose: FSTCW writes the current 8087 control word to the memory location defined by the destination.

Format: FSTCW *destination*

Remarks: The FSTCW form of this instruction is preceded by an assembler-generated WAIT instruction. This is the normal form of this instruction. FNSTCW does not cause a WAIT to be generated.

> **Note:** See FNSTCW and the 8087 control word for related information.

Logic: mem-op ← 8087 control word

Exception Flags: None

Encoding:

```
10011011 11011001 mod111r/m disp-low disp-high
   9B       D9    mod111r/m disp-low disp-high
```

Example:

```
FSTCW CONTROL_CONTROL_WORD   ;STORE CONTROL WORD
```

FSTENV (8087)
Store Environment

Purpose: FSTENV writes the 8087's status (control, status and tag words, and exception pointers) to the memory location defined by the *destination* operand.

Format: FSTENV *destination*

Remarks: The FSTENV form of this instruction is preceded by an assembler-generated WAIT instruction. This ensures that any instruction currently active completes its execution before the environment is stored, and that the environment stored is current. After storing, FSTENV sets all exception masks in the processor; it does not affect the interrupt-enable mask. Use FNSTENV when a 8088 WAIT is not desired.

FSTENV/FNSTENV must be allowed to finish before any other 8087 instruction is decoded. When FSTENV is coded, an assembler-generated WAIT should precede the next 8087 instruction.

> **Note:** See FNSTENV and the 8087 environment for related information.

Logic: mem-op ← 8087 environment

Exception Flags: None

Encoding:

```
10011011 11011001 mod110r/m disp-low disp-high
   9B       D9    mod110r/m disp-low disp-high
```

Example:

```
FSTENV STORE_ENVIRONMENT   ;STORE ENVIRONMENT
```

FSTP (8087)
Store Real and POP

Purpose: FSTP transfers the 8087 stack top element ST to the location defined by the *destination* operand. The 8087 stack top element is then popped from the 8087 stack.

Format: FSTP *destination*

Remarks: The *destination* can be another 8087 stack element or memory operand (short-real, long-real, or temporary-real). If the *destination* is short or long real memory, the significand is rounded to the width of the *destination* according to the RC field of the control word and the exponent is converted to the width and bias of the *destination* format.

FSTP permits storing temporary real numbers in a memory operand while FST does not. FSTP ST(0) is equivalent to popping the 8087 stack with no data transfer.

Note: See FST for related information.

Exception Flags: I,O,U,P

FSTP ST to Register operand:

Format: FSTP ST(*i*)

Remarks: ST to 8087 stack element ST(*i*) and pop 8087 stack.

Logic: ST(*i*) ← ST
pop 8087 stack

Encoding: 10011011 11011101 11011(*i*)
9B DD D8 + (*i*)

FSTP (8087)
Store Real and POP

Example:

```
FSTP   ST(5)    ;STORE REAL AND POP
```

FSTP ST to Memory operand:

Format: FSTP short_real
or
FSTP long_real
or
FSTP temp_real

Remarks: Transfer ST to memory operand and pop 8087 stack.

Encoding: SHORT_REAL
10011011 11011001 mod011r/m disp-low disp-high
9B D9 mod011r/m disp-low disp-high

Example:

```
FSTP   SHORT_REAL       ;STORE REAL AND POP
```

Encoding: LONG_REAL
10011011 11011101 mod011r/m disp-low disp-high
9B DD mod011r/m disp-low disp-high

Example:

```
FSTP   LONG_REAL        ;STORE REAL AND POP
```

Encoding: TEMPORARY_REAL
10011011 11011011 mod111r/m disp-low disp-high
9B DB mod111r/m disp-low disp-high

Example:

```
FSTP   TEMPORARY_REAL   ;STORE REAL AND POP
```

FSTSW (8087)
Store Status Word

Purpose: FSTSW writes the current value of the 8087 status word to the memory location defined by the *destination* operand.

Format: FSTSW *destination*

Remarks: The FSTSW form of this instruction is preceded by an assembler-generated WAIT instruction. In special cases where this WAIT is not desired, use the FNSTSW instruction.

This is a useful instruction to trigger various conditional branching routines by keying on the state of the fields in the status word.

If the WAIT form (FSTSW) is used with an outstanding unmasked exception, deadlock results.

> **Note:** See FNSTSW and the 8087 status word for related information.

Logic: mem-op ← 8087 status word

Exception Flags: None

Encoding: 10011011 11011101 mod111r/m disp-low disp-high
9B DD mod111r/m disp-low disp-high

Example:

```
FSTSW STORE_STATUS_WORD   ;STORE STATUS WORD
```

FSUB (8087)
Subtract Real

Purpose: FSUB subtracts the *source* operand from the *destination* operand and returns the difference to the destination.

Format: FSUB (no operands)
or
FSUB *source*
or
FSUB *destination,source*

Remarks: The difference of the two operands is stored in the *destination* (leftmost) operand. FSUB can be written without operands, with only a *source* or with a *destination* and a source.

Note: See FSUBP and FISUB for related information.

Exception Flags: I,D,O,U,P

FSUB (no operands) 8087 stack form:

Format: FSUB [ST(1),ST]

Note: ST(1),ST are the implied operands; they are not coded, and are shown here for information only.

Remarks: FSUB picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

Note: FSUB (no operands) is similar to the FSUBP instruction with ST(*i*) being ST(1).

FSUB (8087)
Subtract Real

Logic: ST(1) ← ST(1) - ST
pop 8087 stack

Encoding: 10011011 11011110 11101001
9B DE E9

Example:

```
FSUB                    ;SUBTRACT REAL
```

FSUB (source) Real memory form:

Format: FSUB short__real
or
FSUB long__real

Remarks: The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this form of the instruction.

> **Note:** Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← ST - mem-op

Encoding: SHORT__REAL
10011011 11011000 mod100r/m disp-low disp-high
9B D8 mod100r/m disp-low disp-high

Example:

```
FSUB   SHORT_REAL       ;SUBTRACT REAL
```

Encoding: LONG__REAL
10011011 11011100 mod100r/m disp-low disp-high
9B DC mod100r/m disp-low disp-high

FSUB (8087)
Subtract Real

Example:

```
FSUB   LONG_REAL          ;SUBTRACT REAL
```

FSUB (destination,source) Register form:

Remarks: Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format: FSUB ST,ST(*i*)

Logic: ST ← ST - ST(*i*)

Encoding: 10011011 11011000 11100(*i*)
9B D8 E0 + (*i*)

Example:

```
FSUB   ST,ST(1)        ;SUBTRACT REAL
FSUB   ST,ST(7)
```

Format: FSUB ST(*i*),ST

Logic: ST(*i*) ← ST(*i*) - ST

Encoding: 10011011 11011100 11101(*i*)
9B DC E8 + (*i*)

Example:

```
FSUB   ST(1),ST        ;SUBTRACT REAL
FSUB   ST(7),ST
```

FSUBP (8087)
Subtract Real and POP

Purpose: FSUBP subtracts the *source* operand from the *destination* operand and returns the result to the *destination* operand. Then the floating-point 8087 stack is Popped.

Format: FSUBP *destination,source*

Remarks: The difference of the two operands is stored in the *destination* (leftmost) operand. FSUBP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. FSUBP can be used to get the 8087 stack top as the *source* operand,and then discard it by popping the 8087 stack.

See FSUB and FISUB for related information.

> **Note:** FSUB (no operands) is similar to the FSUBP instruction with ST(*i*) being ST(1).

Logic: ST(*i*) ← ST(*i*) - ST
pop 8087 stack

Exception Flags: I,D,O,U,P

Encoding: 10011011 11011110 11101(*i*)
9B DE E8 + (*i*)

Example:

```
FSUBP  ST(7),ST       ;SUBTRACT REAL AND POP
```

FSUBR (8087)
Subtract Real Reversed

Purpose: FSUBR subtracts the *destination* operand from the *source* operand and returns the result to the destination.

Format: FSUBR (no operands)
or
FSUBR *source*
or
FSUBR *destination,source*

Remarks: FSUBR reverses the normal order of operation. All other properties are the same as the FSUB instruction.

The difference of the two operands is stored in the *destination* (leftmost) operand. FSUBR can be written without operands, with only a *source* or with a *destination* and a source.

> **Note:** See FSUB, FSUBP, FSUBRP and FISUB for related information.

Exception Flags: I,D,O,U,P

FSUBR (8087)
Subtract Real Reversed

FSUBR (no operands) 8087 stack form:

Format: FSUBR [ST(1),ST]

> **Note:** ST(1),ST are the implied operands; they are not coded, and are shown here for information only.

Remarks: FSUBR picks the *source* operand from the 8087 stack top and the *destination* operand from the next 8087 stack element. It then pops the 8087 stack, does the operation, and returns the result to the new 8087 stack top.

> **Note:** FSUBR (no operands) is similar to the FSUBRP instruction with ST(*i*) being ST(1).

Logic: ST(1) ← ST - ST(1)
pop 8087 stack

Encoding: 10011011 11011110 11100001
9B DE E1

Example:

```
FSUBR                          ;SUBTRACT REAL
```

FSUBR (8087)
Subtract Real Reversed

FSUBR (source) Real memory form:

Format: FSUBR short__real
or
FSUBR long__real

Remarks: The real memory form permits a real number in memory to be used directly as a *source* operand. The *destination* operand is the top of 8087 stack register ST. It is implied in this form of the instruction.

> Note: Any memory addressing mode can be used to define the *source* operand.

Logic: ST ← mem-op - ST

Encoding: SHORT__REAL
10011011 11011000 mod101r/m disp-low disp-high
9B D8 mod101r/m disp-low disp-high

Example:

```
FSUBR   SHORT_REAL
```

Encoding: LONG__REAL
10011011 11011100 mod101r/m disp-low disp-high
9B DC mod101r/m disp-low disp-high

Example:

```
FSUB   LONG_REAL
```

FSUBR (8087)
Subtract Real Reversed

FSUBR (destination,source) Register form:

Remarks: Specify the 8087 stack top as one operand and any register on the 8087 stack as the other operand.

Format: FSUBR ST,ST(*i*)

Logic: ST ← ST(*i*) - ST

Encoding: 10011011 11011000 11101(*i*)
9B D8 E8 + (*i*)

Example:

```
FSUBR   ST,ST(1)        ;SUBTRACT REAL
FSUBR   ST,ST(7)
```

Format: FSUBR ST(*i*),ST

Logic: ST(*i*) ← ST - ST(*i*)

Encoding: 10011011 11011100 11100(*i*)
9B DC E0 + (*i*)

Example:

```
FSUBR   ST(1),ST
FSUBR   ST(7),ST
```

FSUBRP (8087)
Subtract Real Reversed and POP

Purpose: FSUBRP subtracts the *destination* operand from the *source* operand and returns the result to the destination. The 8087 stack top is then Popped.

Format: FSUBRP *destination,source*

Remarks: FSUBRP reverses the normal order of operation. All other properties are the same as the FSUBP instruction.

The difference of the two operands is stored into the *destination* (leftmost) operand. FSUBRP picks the *source* operand from the ST register, or top of 8087 stack, and the *destination* operand from an ST(*i*) element. It then pops the 8087 stack, does the operation, and returns the result to the ST(*i*) 8087 stack. FSUBRP can be used to get the 8087 stack top as the *source* operand, and then discard it by popping the 8087 stack.

See FSUB, FSUBR, FSUBP, and FISUBR for related information.

Logic: ST(*i*) ← ST - ST(*i*)
pop 8087 stack

Exception Flags: I,D,O,U,P

Encoding: 10011011 11011110 11100(*i*)
9B DE E0 + (*i*)

Example:

```
FSUBRP  ST(7),ST      ;SUBTRACT REAL AND POP
```

FTST (8087)
Test

Purpose: FTST tests the floating-point 8087 stack top element by comparing it to +0.0. The result is posted to the condition code of the status word.

Format: FTST (no operands)

Remarks: The results of FTST are posted to the condition code as follows:

C3	C0	Result
0	0	ST is positive and non-zero
0	1	ST is negative and non-zero
1	0	ST is zero (+ or -)
1	1	ST is not comparable (for example, it is a NAN or projective ∞)

Note: See FCOM, FCOMP, FCOMPP, FICOM, FICOMP, and FXAM for other comparison instructions.

Logic: ST ← ST - 0.0

Exception Flags: I,D

Encoding: 10011011 11011001 11100100
9B D9 E4

Example:

```
FTST                    ;TEST
```

FWAIT (8087)
Wait (CPU Instruction)

Purpose: FWAIT causes the 8088 to wait until the current 8087 instruction is completed before the 8088 executes the next instruction.

Format: FWAIT (No Operands)

Remarks: FWAIT synchronizes the 8087 to the 8088. The FWAIT instruction should *always* be used for this purpose and *never* the 8088 WAIT instruction. Using the 8088 WAIT instruction for this purpose could cause an infinite wait condition in some circumstances.

FWAIT is an alternate mnemonic for the 8088 WAIT instruction. FWAIT and WAIT assemble with the same object code when using the /R assembler parameter and different object code when using the /E assembler parameter.

> **Note:** FWAIT must be used instead of WAIT if 8087 emulator compatibility is desired.

The routines that change an object program for 8087 emulation change any FWAITS to NOPs but do not change any WAITs. The program waits forever if a WAIT is encountered in emulated execution since there is no 8087 to drive the 8088's test pin active.

FWAIT (8087)
Wait (CPU Instruction)

Logic: 8088 wait

Exception Flags: None

Encoding: 10011011
9B

Example: This example shows how the FWAIT instruction can be used to force the CPU to WAIT until the variable STATUS has been updated by the 8087 instruction FNSTSW to insure that the information in STATUS is the most recently updated information.

```
FNSTSW      STATUS
FWAIT       ;Wait for FNSTSW
MOV         AX,STATUS
```

FXAM (8087)
Examine

Purpose: FXAM examines the contents of the 8087 stack top element (ST), and posts the result in the condition code field of the status word.

Format: FXAM

Remarks: The result of FXAM is posted to the condition code field of the status word as follows:

C3	C2	C1	C0	**Interpretation**
0	0	0	0	+ Unnormal
0	0	0	1	+ NAN
0	0	1	0	- Unnormal
0	0	1	1	- NAN
0	1	0	0	+ Normal
0	1	0	1	+ ∞
0	1	1	0	- Normal
0	1	1	1	- ∞
1	0	0	0	+ 0
1	0	0	1	Empty
1	0	1	0	- 0
1	0	1	1	Empty
1	1	0	0	+ Denormal
1	1	0	1	Empty
1	1	1	0	- Denormal
1	1	1	1	Empty

Note: Although four different encodings can be returned for an empty register, bits C3 and C0 of the condition code are both 1 in all encodings. Bits C2 and C1 should be ignored when examining for empty.

Exception Flags: None

Encoding:
```
10011011 11011001 11100101
   9B       D9       E5
```

Example:

```
FXAM                ;EXAMINE
```

FXCH (8087)
Exchange Registers

Purpose: FXCH swaps the contents of the *destination* and the 8087 stack top registers. If the *destination* is not coded explicitly, ST(1) is used.

Format: FXCH (No Operands)
or
FXCH *destination*

Remarks: Many 8087 instructions operate only on the 8087 stack top. FXCH provides a way of effectively using these instructions on the lower 8087 stack elements.

Exception Flags: I

FXCH (No Operands):

Format: FXCH (No Operands)

Logic:
$T_1 \leftarrow ST(1)$
$ST(1) \leftarrow ST$
$ST \leftarrow T_1$

Encoding:
```
10011011 11011001 11001000
   9B       D9       C8
```

FXCH (8087)
Exchange Registers

FXCH (destination):

Format: FXCH *destination*

Logic:

$T_1 \leftarrow ST(i)$
$ST(i) \leftarrow ST$
$ST \leftarrow T_1$

Encoding: 10011011 11011001 11001(*i*)
9B D9 C8 + (*i*)

Example:

```
FXCH   ST(3)    ;EXCHANGE REGISTERS
FSQRT
FXCH   ST(3)
```

This example takes the square root of the the third register from the top of the 8087 stack.

FXTRACT (8087)

Extract Exponent and Significand

Purpose: FXTRACT factors the number in the 8087 stack top into a significand and an exponent expressed in real numbers. The exponent replaces the original operand in the 8087 stack top, then the significand is pushed onto the 8087 stack.

Format: FXTRACT (no operands)

Remarks: FXTRACT is useful with FBSTP for converting numbers in 8087 temporary real format to decimal representations (for example, for printing or displaying). It can also be useful for debugging, because it allows the exponent and significand parts of a real number to be examined separately.

> **Note:** Numbers are stored internally in a biased exponent format. In this format a true zero exponent is expressed as 16,383 or 3FFFH. FXTRACT converts this biased notation into true notation.

Logic:

```
$T_1 \leftarrow$ exponent(ST)
$T_2 \leftarrow$ significand(ST)
$ST \leftarrow T_1$
push 8087 stack
$ST \leftarrow T_2$
```

Exception Flags: I

Encoding:

```
10011011 11011001 11110100
   9B       D9       F4
```

Example:

```
FXTRACT   ;EXTRACT EXPONENT AND SIGNIFICAND
```

FYL2X (8087)
$Y*Log_2X$

Purpose: FYL2X calculates the $Z=Y*\log_2 X$ function.

Format: FYL2X (no operands)

Remarks: X is taken from the 8087 stack top and Y from ST(1). FYL2X pops the 8087 stack and returns Z to the new 8087 stack top, replacing the Y operand.

The following function optimizes the calculation of Log to any base other than 2 since a multiplication is always required.

$\text{Log}_n X = (1 \div (\text{Log}_2 n)) * \text{Log}_2 X$

Note: The operands must be in the ranges:
$0 < X < \infty$
$-\infty < Y < +\infty$

Logic:

```
$T_1 \leftarrow ST(1) * Log_2(ST)$
pop 8087 stack
$ST \leftarrow T_1$
```

Exception Flags: P (operands not checked)

Encoding: 10011011 11011001 11110001
9B D9 F1

Example:

```
FYL2X              ;Y TIMES LOG BASE 2 OF X
```

FYL2XP1 (8087)
$Y*Log_2(X+1)$

Purpose: FYL2XP1 calculates the $Z=Y*Log_2(X+1)$ function.

Format: FYL2XP1 (no operands)

Remarks:

The value X is taken from the 8087 stack top and must be in the range $0 < |X| < (1-\sqrt{2}/2)$. The value Y is taken from ST(1) and must be in the range $-\infty < Y < +\infty$. FYL2XP1 pops the 8087 stack and returns Z at the new 8087 stack top, replacing Y.

This instruction provides improved accuracy over FYL2X when computing the Log of a number very close to 1.

Note: See FYL2X for related information.

Logic:

```
$T_1 \leftarrow ST + 1$
$T_2 \leftarrow ST(1) * Log_2T_1$
pop 8087 stack
$ST \leftarrow T_2$
```

Exception Flags: P (operands not checked)

Encoding: 10011011 11011001 11111001
9B D9 F9

Example:

```
FYL2XP1          ;Y TIMES LOG BASE 2 OF (X+1)
```

HLT
Halt

Purpose:	The HLT instruction causes the processor to enter its halt state.
Format:	HLT (no operands)
Remarks:	The halt state is cleared by an enabled external interrupt or reset.
Flags:	None
Encoding:	11110100 F4

Example:

```
HLT
```

IDIV
Integer Division, Signed

Purpose: IDIV does a signed division of the double-length NUMR operand, contained in the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation) by the DIVR operand, contained in the specified *source* operand. It returns the single-length quotient (QUO operand) to the accumulator (AL or AX), and returns the single-length remainder (the REM operand) to the accumulator extension (AH for 8-bit operation or DX for 16-bit operation).

Format: IDIV *source*

Remarks: If the quotient is positive and greater than MAX or if the quotient is negative and less than (0-MAX-1), (as when division by zero is attempted) then QUO and REM are undefined, and a type 0 interrupt is generated. IDIV truncates non-integral quotients and returns a remainder with the same sign as the numerator.

If the division results in a value larger than can be held by the appropriate registers, an interrupt of type 0 is generated. The flags are pushed onto the stack, IF and TF are reset to 0, and the CS register contents are pushed onto the stack. CS is then filled by the word at location 2. The current IP is pushed onto the stack and IP is then filled with the word at 0. This sequence thus includes a FAR call to the interrupt handling procedure whose segment and offset are stored at locations 2 and 0.

If the division result can fit in the appropriate registers, the quotient is stored in AL or AX (for word operands) and the remainder in AH or DX, respectively.

IDIV
Integer Division, Signed

Logic:

```
(temp) ← NUMR
if (temp)/(DIVR) > 0
  and (temp)/(DIVR) > MAX
or (temp)/(DIVR) < 0
  and (temp)/(DIVR) < 0-MAX-1
    then
      (QUO),(REM) undefined
      (SP) ← (SP) - 2
      ((SP)+1:(SP)) ← FLAGS
      (IF) ← 0
      (TF) ← 0
      (SP) ← (SP) - 2
      ((SP)+1:(SP)) ← CS
      (CS) ← (2)
      (SP) ← (SP) - 2
      ((SP)+1:(SP)) ← (IP)
      (IP) ← 0
else
      (QUO) ← (temp)/(DIVR)
      (REM) ← (temp)%(DIVR)
```

Flags: Affected— No valid flags result
Undefined— AF, CF, OF, PF, SF, ZF

Encoding: 1111011w mod111r/m
F6 + w mod111r/m

If w=0, NUMR=AX, DIVR=EA, QUO=AL,
REM=AH, MAX=7FH.
If w=1, NUMR=DX:AX, DIVR=EA, QUO=AX,
REM=DX, MAX=7FFFH.

IDIV
Integer Division, Signed

Example: **To divide a word by a byte:**

```
MOV   AX,NUMERATOR_WORD[BX]
IDIV  DIVISOR_BYTE[BX]
```

To divide a word by a word:

```
MOV   AX,NUMERATOR_WORD
CWD   ;CONVERTS WORD TO DOUBLEWORD
IDIV  DIVISOR_WORD
```

To divide a doubleword by a word:

```
MOV   DX,NUM_HI_WORD
MOV   AX,NUM_LO_WORD
IDIV  DIVISOR_WORD[SI]
```

(SEE DIV instruction)

IMUL
Integer Multiply

Purpose: IMUL does a signed multiplication of the accumulator (AL or AX) and the *source* operand, returning a double-length result to the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is not the sign-extension of the low half of the result.

Format: IMUL *source*

Remarks: The accumulator (AL if byte, AX if word) is multiplied by the specified operand. If the high-order half of the result is the sign-extension of the low-order half, the carry and overflow flags are reset, otherwise they are set.

Logic:

```
(DEST) ← (LSRC)*(RSRC)
  where * is signed multiply
if (EXT)=sign-extension of (LOW) then
  (CF) ← 0
else (CF) ← 1
    (OF) ← (CF)
```

Flags: Affected— CF, OF
Undefined—AF, PF, SF, ZF

Encoding: 1111011w mod101r/m
F6 + w mod101r/m

If w=0, LSRC=AL, RSRC=EA, DEST=AX,
EXT=AH, LOW=AL.
If w=1, LSRC=AX, RSRC=EA, DEST=DX:AX

IMUL
Integer Multiply

Example: Any of the memory operands, in the following examples could be an indexed address-expression of the correct TYPE. LSRC__BYTE could be ARRAY[SI] if ARRAY were of type BYTE, and RSRC__WORD could be TABLE[BX][DI] if TABLE were of type WORD.

To multiply a byte by a byte:

```
MOV   AL,LSRC_BYTE
IMUL  RSRC_BYTE ;RESULT IN AX
```

To multiply a word by a word:

```
MOV   AX,LSRC_WORD
IMUL  RSRC_WORD
;HIGH-HALF RESULT IN DX, LOW-HALF IN AX
```

To multiply a byte by a word:

```
MOV   AL,MUL_BYTE
CBW   ;CONVERTS BYTE IN AL TO WORD IN AX
IMUL  RSRC_WORD
;HIGH-HALF RESULT IN DX, LOW-HALF IN AX
```

IMUL (80286)
Integer Immediate Multiply

Purpose: IMUL immediate does a signed multiplication of a value by an immediate value, allowing the specification of a destination register other than the combination of DX and AX.

Format: IMUL *destination,immediate*
or
IMUL *destination,source,immediate*

Remarks: IMUL immediate requires three arguments:

- The immediate value
- The effective address of the second operand
- The register in which the result is to be placed

The two operands are the immediate value, and the data at an effective address (which may be the same register in which the result is placed, another register, or a memory location).

If the immediate operand is a byte, the processor sign extends it to 16 bits before doing the multiplication. The result must always be placed in one of the general purpose registers. The result cannot exceed 16 bits without causing an overflow, and only the lower 16 bits of the result is saved.

IMUL Immediate can also be used with unsigned operands because the low 16 bits of a signed or unsigned multiplication of two 16-bit values is always the same.

If the high-order half of the result is the sign-extension of the low-order half, the carry and overflow flags are reset, otherwise they are set.

IMUL (80286)
Integer Immediate Multiply

Logic:

```
(DEST)←(LSRC)*(RSRC)
  where * is signed multiply
if (EXT)=sign-extension of (LOW)
  then (CF)←0
else (CF)←1
(OF)←(CF)
```

Flags:

Affected— CF, OF
Undefined—AF, PF, SF, ZF

Encoding:

011010s1 data [data if s=0]
69 + s data [data if s=0]

Example: In this example IMUL replaces the contents of BX with the product of the contents of SI and an immediate value of 7.

```
IMUL BX,SI,7
```

In this example IMUL replaces the contents of BX with the product of the contents of BX and an immediate value of 7.

```
IMUL BX,7
```

IN
Input Byte or Word

Purpose: The contents of the *accumulator* are replaced by the contents of the designated *port.*

Format: IN *accumulator,port*

Remarks: IN transfers a byte or word from an input *port* to the AL register (or AX register). The *port* is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a *port* number in the DX register, allowing variable access to 64K input ports.

The *destination* for input must be AX or AL, and must be specified in order for the assembler to know the type of the input. The port names must be immediate values between 0 and 255, as used in the following example or literally. The register name is DX, which must be filled earlier with the requisite port location.

Logic: (DEST) ← (SRC)

Flags: None.

IN

Input Byte or Word

Fixed Port:

Encoding: 1110010w port
E4 + w port

If w=0, SRC=port, DEST=AL.
If w=1, SRC=port+1, DEST=AX.

Example:

```
IN AX,WORD_PORT   ;INPUT WORD TO AX
IN AL,BYTE_PORT   ;INPUT A BYTE TO AL
```

Variable Port:

Encoding: 1110110w
EC + w

If w=0, SRC=(DX), DEST=AL.
If w=1, SRC=(DX)+1:(DX), DEST=AX.

Example:

```
IN AX,DX ;INPUT A WORD TO AX
IN AL,DX ;INPUT A BYTE TO AL
```

INC
Increment Destination by One

Purpose: INC does an addition of the operand and 1, and returns the result to the operand.

Format: INC *destination*

Remarks: The specified operand is incremented by 1. There is no carry-out of the most-significant bit.

Logic: (DEST) ← (DEST) + 1

Flags: Affected— AF, OF, PF, SF, ZF

Register Operand (Word):

Encoding: 01000reg
40 + reg

```
DEST=REG
```

Example:

```
INC AX
INC DI
```

INC
Increment Destination by One

Memory or Register Operand:

Encoding: 1111111w mod000r/m
FE + w mod000r/m

```
DEST=EA
```

Example: **Increment register:**

```
INC CX
INC BL
```

Increment memory:

```
INC   MEM_BYTE
INC   MEM_WORD[BX]
INC   BYTE PTR[BX]
;BYTE IN DATA SEGMENT AT OFFSET [BX]
INC   ALPHA[DI][BX]
INC   BYTE PTR [SI][BP]
;BYTE IN STACK SEGMENT AT OFF [SI + BP]
INC   WORD PTR [BX]
;INCREMENTS THE WORD IN DATA SEGMENT AT
; OFFSET [BX]
```

INS/INSB/INSW (80286)
Input from Port to String

Purpose: INS transfers a byte or word string element from an input *port* numbered by the DX register to the memory at ES:DI. The type of the first operand to INS determines whether a byte or a word is moved.

Format: INS *destination-string,port*
or
INSB (no operands)
or
INSW (no operands)

Remarks: The address of the destination is determined only by the contents of DI, not by the first operand to INS. The correct index value must be loaded into DI before executing the INS, INSB, or the INSW instructions. The operand is used only to validate ES segment addressability and to determine the data type. The operand must be addressable from the ES register; no segment override is possible.

The port must be addressed through the DX register; the port number cannot be specified as an immediate value.

INSB and INSW are synonyms for the byte and word INS instructions. They are simpler to use, requiring no operands, however no type or segment checking is done.

INS/INSB/INSW (80286)

Input from Port to String

DI is advanced after the transfer is done. If the direction flag is 0, DI increments; if the direction flag is 1, DI decrements. DI is altered by 1 if a byte was moved; by 2 if a word was moved.

INS, INSB, and INSW can be preceded by the REP prefix for a block input of CX bytes or words. See the REP instruction for the details of this operation.

> **Note:** Not all input port devices can handle the speed at which this instruction executes.

Logic: (DEST)←(SRC)

Flags: None.

Encoding: 0110110w
6C + w

If w=0, DEST=byte
If w=1, DEST=word

Example:

```
INS BSTRING,DX   ;INPUT BYTE
INS WSTRING,DX   ;INPUT A WORD
INSB             ;INPUT BYTE
INSW             ;INPUT A WORD
```

INT
Interrupt

Purpose: INT pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through any one of the 256 vector elements. The 1-byte form of this instruction generates a type 3 interrupt.

Format: INT *interrupt-type*

Remarks: Stack Pointer is decremented by 2 and all flags are pushed into the stack. The interrupt and trap flags are then reset. SP is then decremented by 2 and the current contents of the CS register are pushed onto the stack. CS is then filled with the high-order word of the doubleword interrupt vector (the segment base-address of the interrupt handling procedure for this *interrupt type*).

SP is then decremented by 2 and the current contents of the Instruction Pointer are pushed onto the stack. IP is then filled with the low-order word of the interrupt vector, located at absolute address TYPE*4. This completes an inter-segment FAR call to the procedure that is to process this *interrupt-type*.

See PUSHF, INTO, and IRET instructions.

INT

Interrupt

Logic:

(SP) ← (SP) - 2
((SP)+1:(SP)) ← FLAGS
(IF) ← 0
(TF) ← 0
(SP) ← (SP) - 2
((SP)+1:(SP) ← (CS)
(CS) ← (TYPE*4 + 2)
(SP) ← (SP) - 2
((SP)+1:(SP)) ← (IP)
(IP) ← (TYPE*4)

Flags: Affected— IF, TF

Encoding:

1100110v type
CC + v type

If v=0, type = 3.
If v=1, type = TYPE.

> **Note:** The operand must be immediate data, not a register or a memory reference.

Example:

```
INT    3   ;ONE BYTE:(11001100)
INT    2   ;TWO BYTES:(11001101 00000010)
INT    67  ;TWO BYTES:(11001101 01000011)
IMM_44 EQU 44
INT    IMM_44 ;TWO BYTES:(11001101 00101100)
```

INTO
Interrupt If Overflow

Purpose: INTO pushes the flag registers (as in PUSHF), clears the TF and IF flags, and transfers control with an indirect call through vector element 4 (location 10H) if the OF flag is set (trap on overflow). If the OF flag is clear, no operation takes place.

Format: INTO (no operands)

Remarks: If the OF is zero, no operation occurs. If OF is 1, the Stack Pointer is decremented by 2 and all flags are saved onto the stack. The trap and interrupt flags are reset. SP is again decremented by 2 and the contents of CS are pushed into the stack. CS is then filled with the second word (segment) of the doubleword interrupt vector for a type 4 interrupt.

SP is again decremented by 2, and the current Instruction Pointer (pointing to the next instruction after INTO) is pushed onto the stack. IP is then filled with the first word of the type 4 doubleword interrupt vector, located at absolute location 16 (10H). This word is the offset of the procedure to handle type 4 interrupts. The segment base address was already placed in CS. This completes a FAR call to the proper procedure.

INTO
Interrupt If Overflow

Logic:

```
if (OF) ← 1 then
   (SP) ← (SP) - 2
   ((SP)+1:(SP)) ← FLAGS
   (IF) ← 0
   (TF) ← 0
   (SP) ← (SP) - 2
   ((SP)+1:(SP)) ← (CS)
   (CS) ← (12H)
   (SP) ← (SP) - 2
   ((SP)+1:(SP)) ← (IP)
   (IP) ← (10H)
```

Flags: IF=0, TF=0

Encoding: 11001110
CE

Example:

```
INTO
```

IRET
Interrupt Return

Purpose: IRET transfers control to the return address saved by a previous interrupt operation and restores the saved flag registers (as in POPF).

Format: IRET (no operands)

Remarks: The instruction Pointer is filled with the word at the top of the stack. The Stack Pointer is then incremented by 2, and the CS register is filled with the word now at the top of the stack. This returns control to the point where the interrupt was encountered.

SP is again incremented by 2, and the flags are restored from the appropriate bits of the word now at the top of the stack. (See the POPF instruction in this chapter.) SP is again incremented by 2.

Logic:

```
(IP) ← ((SP)+1:(SP))
(SP) ← (SP) + 2
(CS) ← ((SP)+1:(SP))
(SP) ← (SP) + 2
FLAGS ← ((SP)+1:(SP))
(SP) ← (SP)+2
```

Flags: Affected— All

Encoding: 11001111
CF

Example:

```
IRET
```

J(condition)
Jump Short If Condition Met

Purpose: J(condition) transfers control to the target operand. Conditional short jumps (except for JCXZ) test the flags. JCXZ tests the contents of the CX register for zero, rather than interrogating the flags.

Format: J(condition) *short-label*

Remarks: If the condition is true, then a short jump is made to the label provided as the operand. The condition for each mnemonic is given in the following table along with the corresponding encoding and description.

The target label must be within −128 to +127 bytes of the next instruction. This range is necessary for the assembler to construct a 1-byte signed displacement from the beginning of the next instruction. If the label is out-of range, or if the label is a FAR label, then a jump with the opposite condition around an unconditional jump to the non-short label must be performed.

> **Note:** "Above" and "below" refer to the relation between two unsigned values. "Greater" and "less" refer to the relation between two signed values.

Logic: If Condition Met then
  (IP) ← (IP) + disp (sign-extended to 16-bits)

> **Note:** See the following table for Jump Short instructions and their Conditions to be tested.

J(condition)

Jump Short If Condition Met

Inst	Jump short if	Condition	Encode
JA	above	CF=0 and ZF=0	77 disp
JAE	above or equal	CF=0	73 disp
JB	below	CF=1	72 disp
JBE	below or equal	CF=1 or ZF=1	76 disp
JC	carry	CF=1	72 disp
JCXZ	CX register is zero	(CF or ZF)=0	E3 disp
JE	equal	ZF=1	74 disp
JG	greater	ZF=0 and SF=OF	7F disp
JGE	greater or equal	SF=OF	7D disp
JL	less	(SF xor OF)=1	7C disp
JLE	less or equal	((SF xor OF) or ZF)=1	7E disp
JNA	not above	CF=1 or ZF=1	76 disp
JNAE	not above nor equal	CF=1	72 disp
JNB	not below	CF=0	73 disp
JNBE	not below nor equal	CF=0 and ZF=0	77 disp
JNC	not carry	CF=0	73 disp
JNE	not equal	ZF=0	75 disp
JNG	not greater	((SF xor OF) or ZF)=1	7E disp
JNGE	not greater nor equal	(SF xor OF)=1	7C disp
JNL	not less	SF=OF	7D disp
JNLE	not less nor equal	ZF=0 and SF=OF	7F disp
JNO	not overflow	OF=0	71 disp
JNP	not parity	PF=0	7B disp
JNS	not sign	SF=0	79 disp
JNZ	not zero	ZF=0	75 disp
JO	overflow	OF=1	70 disp
JP	parity	PF=1	7A disp
JPE	parity even	PF=1	7A disp
JPO	parity odd	PF=0	7B disp
JS	sign	SF=1	78 disp
JZ	zero	ZF=1	74 disp

J(condition)
Jump Short If Condition Met

Flags: None

Encoding: 01110111 disp (for JA/JNBE)
77 disp (for JA/JNBE)

Note: See the previous table for other Jump Short instructions and their encodings.

Example:

```
JA   TARGET_LABEL
JNBE TARGET_LABEL
```

JMP
Jump

Purpose: JMP transfers control to the *target* operand.

Format: JMP *target*

Remarks: The jump is always relative to the segment base address in the CS register. A direct jump uses the offset (and segment, if FAR) byte that follows the instruction byte. Indirect jumps use the contents of the location addressed by the bytes that follow the instruction byte.

The Instruction Pointer is replaced by the *target's* offset in all inter-segment jumps, and in intra-segment (or intra-group) indirect jumps.

When the jump is a direct intra-segment or intra-group, the distance from the end of the instruction to the *target* label is added to the IP.

Inter-segment jumps first replace the contents of CS, using the second word following the instructions (direct) or using the second word following the indicated data address (indirect).

Logic: If inter-segment then (CS) ← SEG
(IP) ← DEST

Flags: None

JMP
Jump

Intra-Segment or Intra-Group Direct:

Encoding: 11101001 disp-low disp-high
E9 disp-low disp-high

DEST=(IP) + disp

Example:

```
JMP    NEAR_LABEL
```

Intra-Segment Direct Short:

Encoding: 11101011 disp
EB disp

DEST=(IP) + disp (sign extended to 16 bits)

Example:

```
JMP    TARGET_LABEL
JMP    SHORT NEAR_LABEL
```

Note: The target label must be within –128 to +127 bytes of the next instruction.

Inter-Segment Direct:

Encoding: 11101010 offset segment
EA offset segment

DEST=offset, SEG=seg

Example:

```
JMP    LABEL_DECLARED_FAR
JMP    FAR PTR LABEL_NAME
JMP    FAR PTR NEAR_LABEL
```

Inter-Segment Indirect:

Encoding: 11111111 mod101r/m
FF mod101r/m

DEST=(EA), SEG=(EA+2)

Example:

```
JMP   VAR_DOUBLEWORD
JMP   DWORD PTR [BX][SI]
JMP   ALPHA [BP][DI]
```

Intra-Segment or Intra-Group Indirect:

Encoding: 11111111 mod100r/m
FF mod100r/m

DEST=(EA)

Example:

```
JMP   TABLE[BX]
JMP   WORDPTR [BX][DI]
JMP   BETA_WORD
JMP   AX
JMP   SI
JMP   BP
```

Note: These last three replace the Instruction Pointer by the contents of the named register. This causes a jump directly to the byte with that offset past CS. This is different from the direct intra-segment jumps, which are self-relative, causing an add to the IP.

LAHF
Load AH from Flags

Purpose: LAHF transfers the flag registers SF, ZF, AF, PF, and CF into specific bits of the AH register. The bits indicated "X" are unspecified.

Format: LAHF (no operands)

Remarks: Specific bits of AH are filled from the following flags: The sign flag fills bit 7. The zero flag fills bit 6. The auxiliary carry flag fills bit 4. The parity flag fills bit 2. The carry flag fills bit 0. Bits 1, 3, and 5 of AH are indeterminate, on some occasions 1 and at other times 0.

Logic: (AH) ← (SF):(ZF):X:(AF):X:(PF):X:(CF)

Flags: None

Encoding: 10011111
9F

Example:

```
LAHF
```

LDS
Load Data Segment Register

Purpose: LDS transfers a "pointer-object" (a 32-bit object containing an offset address and a segment address) from the DWORD *source* operand (which must be a doubleword memory operand) to a pair of *destination* registers. The segment address is transferred to the DS segment register. The offset address can be transferred to any 16-bit general, pointer, or index register you specify (not a segment register).

Format: LDS *destination,source*

Remarks: The contents of the specified register is replaced by the lower addressed word of the doubleword memory operand.

The contents of the DS register are replaced by the higher-addressed word of the doubleword memory operand.

Logic: (REG) ← (EA)
(DS) ← (EA + 2)

Flags: None

Encoding: 11000101 modregr/m
C5 modregr/m

For mod ≠ 11.
If mod = 11, undefined operation.

Example:

```
LDS   BX,ADDR_TABLE[SI]
LDS   SI,DWORD PTR NEWSEG[BX]
```

LEA
Load Effective Address

Purpose: LEA transfers the offset address of the source operand to the destination operand. The contents of the specified register are replaced by the offset of the indicated variable, label or address-expression.

Format: LEA *destination,source*

Remarks: The *source* operand must be a symbol defining a memory location and the *destination* operand can be any 16-bit general, pointer, or index register.

LEA allows the *source* to be subscripted. This is not allowed using the MOV instruction with the OFFSET operator. Also, the later operation uses the offset of the variable in the segment where it was defined. LEA, however, takes into account a group offset if the group is the only possible access route via the latest ASSUME pseudo-op (see Chapter 2).

Logic: (REG) ← EA

Flags: None

Encoding: 10001101 modregr/m
8D modregr/m

For mod ≠ 11.
If mod = 11, undefined operation.

Example:

```
LEA   BX,VARIABLE_7
LEA   DX,BETA[BX][S1]
LEA   AX,[BP][DI]
```

LEAVE (80286)
High Level Procedure Exit

Purpose: LEAVE executes a procedure return for a high level language.

Format: LEAVE (no operands)

Remarks: LEAVE is the complementary operation to ENTER; it reverses the effects of that instruction. The LEAVE instruction deallocates all local variables and restores the SP and BP registers to their values immediately after the procedure's invocation.

LEAVE releases the stack space used by a procedure by copying BP to SP. The old frame pointer is now popped into BP, restoring the caller's frame, and a subsequent RET *nn* instruction will follow the back-link and remove any arguments pushed on the stack for the exiting procedure.

Do not confuse LEAVE with the $LEAVE Structure Statement. See Chapter 4 "SALUT" for Structure Statements.

Logic:

$(SP) \leftarrow (BP)$
$(BP) \leftarrow ((SP)+1:(SP))$
$(SP) \leftarrow (SP)+2$

Flags:

Affected— None
Undefined— None

Encoding:

11001001
C9

Example:

```
LEAVE
```

LES
Load Extra Segment Register

Purpose: LES transfers a "pointer object" (a 32-bit object containing an offset address and a segment address) from the DWORD *source* operand (which must be a doubleword memory operand), to a pair of *destination* registers. The segment address is transferred to the ES segment register. The offset address can be transferred to a 16-bit general, pointer, or index register (not a segment register).

Format: LES *destination,source*

Remarks: The contents of the specified register are replaced by the lower addressed word of the doubleword memory operand. The contents of the ES register are replaced by the higher addressed word of the doubleword memory operand.

Logic:
(REG) ← (EA)
(ES) ← (EA + 2)

Flags: None

Encoding:
11000100 modregr/m
C4 modregr/m

For mod ≠ 11.
If mod = 11, undefined operation.

Example:

```
LES   BX,ADDR_TABLE[SI]
LES   DI,DWORD PTR NEWSEG[BX]
```

LOCK
Lock Bus

Purpose: A special 1-byte lock prefix can precede any instruction. It causes the processor to assert its bus-lock signal for the duration of the operation caused by the instruction. In multiple processor systems with shared resources, it is necessary to provide mechanisms to enforce controlled access to those resources. Such mechanisms, while generally provided through software operating systems, require hardware assistance. A mechanism for accomplishing this is a *locked exchange* (also called test-and-set-lock).

Format: LOCK (no operands)

Remarks: The instruction that is most useful in this context is an exchange register with memory. You can implement a simple software lock with the following code sequence.

Flags: None

Encoding: 11110000
F0

Example:

```
CHECK: MOV AL,1 ;SET AL TO 1
;                      (IMPLIES LOCKED)
LOCK   XCHG  SEMO,AL ;TEST AND SET LOCK
        TEST  AL,AL  ;SET FLAGS BASED ON AL
        JNZ   CHECK  ;RETRY IF LOCK ALREADY SET
        o
        o
        o
        MOV  SEMO,0  ;CLEAR THE LOCK WHEN DONE
```

LODS/LODSB/LODSW
Load Byte or Word String

Purpose: LODS transfers a byte (or word) operand from the *source* operand addressed by SI to accumulator AL (or AX) and adjusts the SI register by DELTA. This operation ordinarily would not be repeated.

Format:
LODS *source-string*
or
LODSB (no operands)
or
LODSW (no operands)

Remarks: The *source* byte (or word) is loaded into AL (or AX). The Source Index is incremented by 1 (or 2, for word strings) if the Direction Flag is reset; otherwise SI is decremented by 1 (or 2).

Logic:
(DEST) ← (SRC)
If (DF)=0, (SI) ← (SI) + DELTA.
If (DF)=1, (SI) ← (SI) - DELTA.

Flags: None

Encoding:
1010110w
AC + w

If w=0,
SRC = (SI), DEST = AL, DELTA = 1.
If w=1,
SRC = (SI) + 1:(SI), DEST = AX, DELTA = 1.

Example:

```
CLD      ;CLEARS DIRECTION FLAGS SO
;         SI WILL BE INCREMENTED
MOV    SI,OFFSET    BYTE_STRING
LODS   BYTE_STRING     ;SI=SI+1
o
o
o
STD      ;SETS (DF) SO SI
;         WILL BE DECREMENTED
MOV    SI,OFFSET     WORD_STRING
LODS   WORD_STRING     ;SI=SI-2
```

In the above example, DF = 1 implies that the variable WORD__STRING names the last or highest-addressed word in the string. The operand named in the LODS instruction is used only by the assembler to verify type and accessibility using correct segment register contents. LODS really uses only SI to point to the location whose contents are to be loaded into the accumulator, without using the name given in the *source* instruction.

The string instructions are unusual in several aspects:

1. Load SI with the offset of the *source-string*.

2. Load DI with the offset of the *destination-string*.

3. Each can be coded with or without symbolic memory operands.

   - If symbolic operands are coded, the assembler can check the addressability of them for you.

   - References that use hardware defaults should be coded using the operand-less forms (LODSB and LODSW), to avoid the additional pointer information.

   - Never use [BX] or [BP] addressing modes with the string instructions.

4. If the instruction mnemonic is coded without operands, the segment registers are as follows:

   - SI defaults to an offset in the segment addressed by DS.

   - DI is required to be an offset in the segment addressed by ES.

LOOP
Loop Until Count Complete

Purpose: LOOP decrements the CX (count) register by 1 and transfers control to the target operand (*short-label*) if CX is not zero.

Format: LOOP *short-label*

Remarks: The Count register (CX) is decremented by 1. If the new CX is not zero, the distance from the end of this instruction to the target label, *short-label*, is added to the Instruction Pointer, effecting the jump. If CX = 0, no jump occurs.

The target label must be within –128 to +127 byte of the next instruction.

Logic:

```
(CX) ← (CX) - 1
If (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Flags: None

Encoding: 11100010
E2

LOOP
Loop Until Count Complete

Example: The following sequence computes the 16-bit check-sum of a non-null array:

```
       MOV    CX,LENGTH ARRAY
       MOV    AX,0
       MOV    SI,AX
;
NEXT:  ADD    AX,ARRAY[SI]
       ADD    SI,TYPE ARRAY
       LOOP   NEXT
       MOV    CKS,AX
```

Example: In the following example, the instructions from FIB to LL is executed N times and stores into the FIBONACCI array the first N terms of the sequence. (1,1,2,3,5,8,13,21 ...)

```
       MOV    AX,0
       MOV    BX,1
       MOV    CX,N   ;NUMBER OP TERMS
       MOV    DI,AX
;
FIB:   MOV    SI,AX
       ADD    AX,BX
       MOV    BX,SI
       MOV    FIBONACCI[DI],AX
       ADD    DI,TYPE FIBONACCI
;
LL:    LOOP   FIB
```

LOOPE/LOOPZ
Loop If Equal/If Zero

Purpose: LOOPE or LOOPZ decrements the CX register by 1 and transfers if CX is not zero and if the ZF flag is set to 1.

Format:
LOOPE *short-label*
or
LOOPZ *short-label*

Remarks: The Count register (CX) is decremented by 1. If the zero flag is set and (CX) is not yet 0, the distance from the end of this instruction to the target label, *short-label*, is added to the Instruction Pointer, effecting the jump. No jump occurs if (ZF) = 0 or if (CX) = 0.

The target label must be within –128 to +127 bytes of the next instruction.

Logic:

```
(CX) ← (CX) - 1
If (ZF) = 1 and (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Flags: None

Encoding:
11100001 disp
E1 disp

LOOPE/LOOPZ
Loop If Equal/If Zero

Example: The following sequence finds the first non-zero entry in a byte array:

```
          MOV    CX,LENGTH ARRAY
          MOV    SI,-1
NEXT:     INC    SI
          CMP    ARRAY[SI], 0
          LOOPE  NEXT
          JNE    OKENTRY
          .   ;ARRIVE HERE IF COMPLETE
          .   ;  ARRAY IS ZERO
          .
OKENTRY:        ;SI TELLS WHICH ENTRY
;                IS NON-ZERO
```

LOOPNE/LOOPNZ
Loop If Not Equal/If Not Zero

Purpose: LOOPNZ or LOOPNE decrements the CX register by 1 and transfers if CX is not 0 and the ZF flag is cleared.

Format: LOOPNZ *short-label*
or
LOOPNE *short-label*

Remarks: The Count register (CX) is decremented by 1. If the new (CX) is not 0 and the zero flag is reset, the distance from the end of this instruction to the target label, *short-label*, is added to the Instruction Pointer, effecting the jump. If (CX) = 0 or if (ZF) = 1, no jump occurs.

The target label must be within –128 to +127 bytes of the next instruction.

Logic:

```
(CX) ← (CX) - 1
If (ZF) = 0 and (CX) ≠ 0 then
  (IP) ← (IP) + disp (sign-extended to 16-bits)
```

Flags: None

Encoding: 11100000 disp
E0 disp

LOOPNE/LOOPNZ
Loop If Not Equal/If Not Zero

Example: The following sequence computes the sum of 2 byte arrays, each of length N, only up to the point of encountering zero entries in both arrays at the same time. At that point, the expression SI-1 gives the length of the nonzero sum arrays.

```
         MOV      AX,0
         MOV      SI-1
         MOV      CX,N
NOZERO:  INC      SI
         MOV      AL,ARRAY1[SI]
         ADD      AX,ARRAY2[SI]
         MOV      SUM[SI],AX
         LOOPNZ   NOZERO
```

The following sequence searches down a linked list for the last element. This is the element with a zero in the word that normally contains the address of the next element. This word is always located the same number of bytes past each list element's beginning. LINK is the name for that absolute number of bytes, for example:

```
       LINK    EQU 7
       MOV     AX,OFFSET HEAD_OF_LIST
       MOV     CX,1000 ;SEARCH AT MOST 1000
;                       ENTRIES
NEXT:  MOV     BX,AX
       MOV     AX,[BX] + LINK
       CMP     AX,0
       LOOPNE  NEXT
```

MOV
Move

Purpose: MOV copies the *source* operand to the *destination* operand. The *source* operand is not modified.

Format: MOV *destination,source*

Remarks: The seven separate types of move instructions are described below. Each type has multiple uses and encodings depending on the type of data being moved and the location of that data. The assembler generates the correct encoding based on the type and location of data. If the *destination* is a register, the bit shown as "d" is a 1, otherwise a 0. If the type is a word, the bit shown as "w" is a 1, otherwise a 0.

Logic: (DEST) ← (SRC)

Flags: None

TO Memory FROM Accumulator:

Encoding: 1010001w addr-low addr-high
A2 + w addr-low addr-high

If w = 0, SRC = AL, DEST = addr.
If w = 1, SRC = AX, DEST = addr + 1:addr.

Example:

```
MOV   ALPHA_MEM,AX
MOV   GAMMA_BYTE, AL
MOV   CS:DATUM_BYTE,AL
MOV   ES:ARRAY[BX][SI],AX
```

MOV

Move

TO Accumulator FROM Memory:

Encoding: 1010000w addr-low addr-high
A0 + w addr-low addr-high

If w = 0, SRC = addr, DEST = AL.
If w = 1, SRC = addr + 1:addr, DEST = AX.

Example:

```
MOV   AX,BETA_MEM
MOV   AL,GAMMA_BYTE
MOV   AX,ES:ARRAY[BX][SI]
MOV   AL,SS:OTHER_BYTE
```

TO Segment Register FROM Memory-or-Register Operand:

Encoding: 10001110 modregr/m
8E modregr/m

If reg ≠ 01 then SRC = EA, DEST = REG else undefined operation.

Example:

```
MOV   ES,DX
MOV   DS,AX
MOV   SS,BX
MOV   ES,SS:NEW_WORD[DI]
```

Note: CS is invalid as a destination here.

MOV
Move

TO Memory-or-Register FROM Segment Register:

Encoding: 10001100 modregr/m
8C modregr/m

SRC = REG, DEST = EA, (DEST) = (SRC)

Example:

```
MOV    DX,DS
MOV    BX,ES
MOV    ARRAY[BX][SI],SS
MOV    BETA_MEM_WORD,DS
MOV    GAMMA,CS
```

Note: CS *is* valid as a source here.

To Register From Register:

To Register From Memory-or-Register Operand:

To Memory-or-Register Operand From Register:

Encoding: 100010d1 modregr/m addr-low* addr-high*
89 + d modregr/m addr-low* addr-high*

If d = 1, SRC = EA, DEST = REG
else SRC = REG, DEST = EA
If d = 0, SRC = REG, DEST = EA.

*These bytes are omitted in register to register moves, when mod = 1,

```
MOV    CX,DX
```

and also when the address-expression to memory is register-indirect with no variable-name-displacement, for example:

```
MOV    [BX][SI],DX
MOV    AX,[BP][DI]
```

Example: **To register from register:**

```
MOV    AX,BX
MOV    CL,DH
MOV    CX,DI
```

To register from memory or register:

```
MOV    AX,MEM_VALUE
MOV    DX,ARRAY[SI]
MOV    DI,MEM[BX][DI]
```

To memory or register from register:

```
MOV    ARRAY[DI],DX
MOV    MEM_VALUE,AX
MOV    [BX][SI],DI
```

MOV
Move

TO Register FROM Immediate-data:

Encoding: 1011 w reg data data-high*
B + w + reg data data-high*

SRC = data, DEST = REG

*Present only if w = 1

Example:

```
MOV   AX,77
MOV   BX,VALUE_14_IMM
MOV   SI,EQU_VAL_9
MOV   DI,618
```

TO Memory-or-Register Operand FROM Immediate-data:

Encoding: 1100011w mod000r/m data data-high*
C6 + w mod000r/m data data-high*

SRC = data, DEST = EA

*Present only if w = 1

Example:

```
MOV   ARRAY[BX][SI],DATA_4
MOV   MEM_BYTE,IMM_BYTE_3
MOV   BYTE PTR [DI],66
MOV   MEM_WORD,1999
MOV   BX,84
MOV   DS:MEM_WORD[BP],3989
```

MOVS/MOVSB/MOVSW
Move Byte or Word String

Purpose: MOVS transfers a byte (or word) operand from the *source* operand addressed by SI to the *destination* operand addressed by DI, and adjusts the SI and DI registers by DELTA (number of bytes specified by the operand TYPE). As a repeated operation, this provides for moving a string from one location in memory to another.

Format:

MOVS *dest-string,source-string*
or
MOVSB (no operands)
or MOVSW (no operands)

Remarks: The *source* string whose offset is in the Source Index is moved into the location in the extra segment. The offset of the extra segment is in the Destination Index. SI and DI are then both incremented, if the direction flag is zero, or both decremented, if (DF) = 1. (See the CLD and STD instructions.) The increment or decrement is 1 for byte strings, 2 for word strings.

Logic:

```
(DEST) ← (SRC)
If (DF) = 0 then
   (SI) ← (SI) + DELTA
   (DI) ← (DI) + DELTA
else
   (SI) ← (SI) - DELTA
   (DI) ← (DI) - DELTA
```

MOVS/MOVSB/MOVSW

Move Byte or Word String

Flags: None

Encoding: 1010010w
A4 + w

If w = 0, SRC = (SI), DEST = (DI), DELTA = 1.
If w = 1, SRC = (SI) + 1:(SI), DEST = (DI) + 1:(DI),
DELTA = 2

Example: For this example, assume type source = BYTE.

```
SOURCE  DB      17 DUP  ' '
DEST    DB      17 DUP  (?)
        .
        .
        .
        CLD  ;CLEAR DF TO AUTO-INCREMENT
        MOV     SI,OFFSET SOURCE
        MOV     DI,OFFSET DEST
        MOV     CX,LENGTH SOURCE ;PASS NUMBER OF
;                                  BYTES IN SOURCE
REP     MOVS    DEST,SOURCE
        or
REP     MOVS    ES:BYTE PTR[DI],DS:[SI]
        or
REP     MOVSB
```

The above sequence moves the complete *source* string (in any segment that can be reached by current segment registers) into the *destination* locations in the extra segment (the ES register is always used for DI operands in string operations). See also "REP". The operands named in the string operation are used only by the assembler to verify type and accessibility using the current segment register's contents. MOVS really moves the byte pointed at by SI to the byte pointed at by DI in ES, without using the names given in the source MOVS instruction.

MOVS/MOVSB/MOVSW
Move Byte or Word String

The string instructions are unusual in several aspects:

1. Load SI with the offset of the *source-string*.
2. Load DI with the offset of the *destination-string*.
3. Each can be coded with or without symbolic memory operands.
    - If symbolic operands are coded, the assembler can check the addressability of them for you.
    - References that use hardware defaults should be coded using the operand-less forms (MOVSB and MOVSW), to avoid the additional pointer information.
    - Never use [BX] or [BP] addressing modes with the string instructions.
4. If the instruction mnemonic is coded without operands, the segment registers are as follows:
    - SI defaults to an offset in the segment addressed by DS.
    - DI is required to be an offset in the segment addressed by ES.

MUL
Multiply, Unsigned

Purpose: MUL does an unsigned multiplication of the accumulator (AL or AX) and the *source* operand, returning a double-length result to the accumulator and its extension (AL and AH for 8-bit operation, or AX and DX for 16-bit operation). CF and OF are set if the top half of the result is nonzero.

Format: MUL *source*

Remarks: The accumulator (AL if byte, AX if word) is multiplied by the specified operand, *source*. If the high-order half of the result is zero, the carry and overflow flags are reset, otherwise they are set.

Logic:

```
(DEST) ← (LSRC) * (RSRC),
   where * is unsigned multiply
if (EXT) = 0 then (CF) ← 0
else (CF) ← 1;
    (OF) ← (CF)
```

Flags:
Affected— CF, OF
Undefined— AF, PF, SF, ZF

Encoding:
1111011w   mod100r/m
F6 + w   mod100r/m

If w = 0,
LSRC = AL, RSRC = EA, DEST = AX,
EXT = AH.
If w = 1,
LSRC = AX, RSRC = EA, DEST = DX:AX,
EXT = DX.

MUL
Multiply, Unsigned

Example: Any of the following memory operands, could be an indexed address-expression of the correct TYPE. LSRC__BYTE could be ARRAY[SI] if ARRAY were of type BYTE, and RSRC__WORD could be TABLE[BX][DI] if TABLE were of type WORD.

To multiply a byte by a byte:

```
MOV   AL,LSRC_BYTE
MUL   RSRC_BYTE        ;RESULT IN AX
```

To multiply a word by a word:

```
MOV   AX,LSRC_WORD
MUL   RSRC_WORD
;HIGH-HALF RESULT IN DX, LOW-HALF IN AX
```

To multiply a byte by a word:

```
MOV   AL,MUL_BYTE
CBW   ;CONVERTS BYTE IN AL TO WORD IN AX
MUL   RSRC_WORD
```

NEG
Negate, Form Twos Complement

Purpose: NEG does a subtraction of the operand from zero, adds 1, and returns the result to the operand. This forms the twos complement of the specified operand.

Format: NEG *destination*

Remarks: The specified operand, *destination*, is subtracted from all 1's (0FFH for bytes, 0FFFFH for words), 1 is added, and the result stored back in *destination.*

Logic: (EA) ← (SRC) - (EA)
(EA) ← (EA) + 1 (affecting flags)

Flags: Affected— AF, CF, OF, PF, SF, ZF

Encoding: 1111011w mod011r/m
F6 + w mod011r/m

If w=0, SRC = 0FFH.
If w=1, SRC = 0FFFFH.

NEG
Negate, Form Twos Complement

Example: If AL contains 13H (00010011),

```
NEG   AL
```

causes AL to contain: -13H or 0EDH (11101101).

If MEM__BYTE contains 0AFH (10101111),

```
NEG   MEM_BYTE
```

causes MEM__BYTE to contain: -0AFH or 51H (01010001).

If SI contains 2FC3H,

```
NEG   SI
```

causes SI to contain: 0D03DH.

NOP
No Operation

Purpose:	NOP causes no operation. The next sequential instruction is then executed.
Format:	NOP (no operands)
Remarks:	The next sequential instruction is executed.
Flags:	None
Encoding:	10010000 90

Example:

```
NOP
```

NOT
Logical Not

Purpose:	NOT forms the ones complement of (inverts) the operand and returns the result to the operand. Flags are not affected.
Format:	NOT *destination*
Remarks:	The specified operand, *destination*, is subtracted from 0FFH (or 0FFFFH, if a word) and the result is stored back into *destination.*
Logic:	(EA) ← SRC - (EA)
Flags:	None
Encoding:	1111011w mod010r/m F6 + w mod010r/m If w = 0, SRC = 0FFH. If w = 1, SRC = 0FFFFH.

Example: If AH contains 13H (00010011), then

```
NOT AH
```

causes AH to contain 0ECH (11101100).

If MEM__BYTE contains 0AFH (10101111), then

```
NOT MEM_BYTE
```

causes MEM__BYTE to contain 50H (01010000).

If DX contains 2FC3H, then

```
NOT DX
```

causes DX to contain 0D03CH.

OR
Logical Inclusive Or

Purpose: OR does the bit-logical inclusive dis-junction of the two operands and returns the result to one of the operands.

Format: OR *destination,source*

Remarks: Each bit position in the *destination* (leftmost) operand becomes 1, unless it and the corresponding bit position of the *source* (rightmost) operand were both 0. Alternative phrasing: each bit position of the result has a 1 if either operand had a 1 in that position; if both had a 0, that position of the result has a zero. The carry and overflow flags are both reset.

Logic:

(DEST) ← (LSRC) OR (RSRC)
(CF) ← 0
(OF) ← 0

Flags:

Affected— CF, OF, PF, SF, ZF
Undefined— AF

Memory or Register Operand with Register Operand:

Encoding:

000010dw modregr/m
08 + dw modregr/m

If d = 1,
LSRC = REG, RSRC = EA, DEST = REG.
If d = 0,
LSRC = EA, RSRC = REG, DEST = EA.

OR
Logical Inclusive Or

Example: **OR register with register:**

```
OR   AH,BL   ;RESULT IN AH,  BL UNCHANGED
OR   SI,DX   ;RESULT IN SI,  DX UNCHANGED
OR   CX,DI   ;RESULT IN CX,  DI UNCHANGED
```

OR memory with register:

```
OR   AX,MEM_WORD
OR   CL,MEM_BYTE[SI]
OR   SI,ALPHA[BX][DI]
```

OR register with memory:

```
OR   BETA[BX][DI],AX
OR   MEM_BYTE,DH
OR   GAMMA[DI],BX
```

Immediate Operand to Accumulator:

Encoding: 0000110w data
0C + w data

If w = 0,
LSRC = AL, RSRC = data, DEST = AL.
If w = 1,
LSRC = AX, RSRC = data, DEST = AX.

Example: **OR immediate (byte):**

```
OR   AL,11110110B
OR   AL,0F6H
```

OR
Logical Inclusive Or

OR immediate (word):

```
OR   AX,23F6H
OR   AX,75Q
OR   AX,23F6H
```

Immediate Operand to Memory or Register Operand:

Encoding: 1000000w mod001r/m data
80 + w mod001r/m data

LSRC = EA, RSRC = data, DEST = EA

Example: **OR immediate with register:**

```
OR     AH,0F6H
OR     CL,37
OR     DI,23F5H
```

OR immediate with memory:

```
OR     MEM_BYTE,3DH
OR     GAMMA[BX][DI],0FACEH
OR     ALPHA[DI],VAL_EQU_33H
```

Another example:

```
BITMASK  EQU  20H
FLAGS    DB   ?
         o
         o
         o
         OR   FLAGS,BITMASK ;TURN ON FLAG BIT
```

OUT
Output Byte or Word

Purpose: The contents of the designated *port* are replaced by the contents of the *accumulator*.

Format: OUT *port,accumulator*

Remarks: OUT transfers a byte (or word) from the AL register (or AX register) to an output *port*. The *port* is specified either with an inline data byte, allowing fixed access to ports 0 through 255, or with a port number in the DX register, allowing variable access to 64K output ports.

Logic: (DEST) ← (SRC)

Flags: None

Fixed Port:

Encoding: 1110011w port
E6 + w port

If w = 0, SRC = AL, DEST = port.
If w = 1, SRC = AX, DEST = port + 1:port.
(0<port<255)

Example:

```
OUT BYTE_PORT_VAL,AL
;               OUTPUTS A BYTE FROM AL
OUT WORD_PORT_VAL,AX
;               OUTPUTS A WORD FROM AX
OUT 44,AX      ;OUTPUTS A WORD FROM AX
;               THROUGH PORT 44
```

OUT
Output Byte or Word

Variable Port:

Encoding: 1110111w
EE + w

If w = 0, SRC = AL, DEST = (DX).
If w = 1, SRC = AX, DEST = (DX) + 1:(DX).

Example:

```
OUT   DX,AL  ;OUTPUTS A BYTE FROM AL
;             THROUGH VARIABLE PORT IN DX
OUT   DX,AX  ;OUTPUTS A WORD FROM AX
;             THROUGH VARIABLE PORT IN AX
```

OUTS/OUTSB/OUTSW (80286)

Output String to Port

Purpose: OUTS transfers a byte or word string element from memory at DS:SI to the *port* numbered by the DX register. The type of the second operand to OUTS determines whether a byte or a word is moved.

Format: OUTS *port,source-string*
 or
OUTSB (no operands)
 or
OUTSW (no operands)

Remarks: The address of the source data is determined only by the contents of SI, not by the second operand to OUTS. The correct index value must be loaded into SI before executing the OUTS, OUTSB, or the OUTSW instructions. The operand is used only to validate segment addressability and to determine the data type. The segment addressability of the operand determines if a segment override byte will be produced, or the default segment register DS is used.

The port must be addressed through the DX register; the port number cannot be specified as an immediate value.

OUTSB and OUTSW are synonyms for the byte and word OUTS instructions. They are simpler to use, requiring no operands, however no type or segment checking is done.

OUTS/OUTSB/OUTSW (80286)

Output String to Port

SI is advanced after the transfer is done. If the direction flag is 0 (CLD was executed), SI increments; if the direction flag is 1 (STD was executed), SI decrements. SI is altered by 1 if a byte was moved; by 2 if a word was moved.

OUTS, OUTSB, and OUTSW can be preceded by the REP prefix for a block input of CX bytes or words. See the REP instruction for the details of this operation.

> **Note:** Not all output devices can handle the speed at which this instruction executes.

Logic: (DEST) ← (SRC)

Flags: None

Encoding: 0110111w
6E + w

If w=0, SRC=byte
If w=1, SRC=word

Example:

```
OUTS DX,BSTRING  ;OUTPUT BYTE
OUTS DX,WSTRING  ;OUTPUT A WORD
OUTSB            ;OUTPUT BYTE
OUTSW            ;OUTPUT A WORD
```

POP

Pop Word Off Stack to Destination

Purpose: The contents of the *destination* are replaced by the word at the top of the stack. The stack pointer is incremented by 2.

Format: POP *destination*

Remarks: POP transfers a word operand from the stack element addressed by the SP register to the *destination* operand and then increments SP by 2.

There are three separate types of POP instructions, for different *destinations*.

Logic: $(DEST) \leftarrow ((SP)+1:(SP))$
$(SP) \leftarrow (SP) + 2$

Flags: None

Register Operand:

Encoding: 01011reg
58 + reg

DEST = REG

Example:

```
POP   CX ;(01011001)
POP   DX ;(01011010)
```

POP
Pop Word Off Stack to Destination

Segment Register:

Encoding: 000reg111
07 + reg

If reg ≠ 01 then DEST = REG else undefined operation.

Note: POP CS is not valid.

Example:

```
POP   SS;(00010111)
POP   DS;(00011111)
```

Memory or Register Operand:

Encoding: 10001111 mod000r/m
8F mod000r/m

DEST = EA

Example:

```
POP   ALPHA  ;(10001111   00000110)
POP   ALPHA[BX] ;(10001111   10000111)
```

POPA (80286)
Pop All General Registers

Purpose:	POPA restores the eight general purpose registers DI, SI, BP, SP, BX, DX, CX, and AX saved on the stack by PUSHA, except that the SP value is discarded instead of loaded into SP.
Format:	POPA
Remarks:	POPA reverses a previous PUSHA, restoring the general purpose registers to their values before PUSHA was executed.
Logic:	Pop DI,SI,BP,SP,BX,DX,CX,AX
Flags:	None
Encoding:	01100001 61

Example:

```
POPA
```

POPF
Pop Flags Off Stack

Purpose: POPF transfers specific bits of the stack element addressed by the SP register to the flag registers and then increments SP by 2.

Format: POPF (no operands)

Remarks: The flag registers are filled from the appropriate bit positions of the word at the top of the stack:

overflow flag = bit 11
direction flag = bit 10
interrupt flag = bit 9
trap flag = bit 8
sign flag = bit 7
zero flag = bit 6
auxiliary carry flag = bit 4
parity flag = bit 2
carry flag = bit 0

Then the stack pointer is incremented by 2.

Logic: $\text{Flags} \leftarrow ((SP)+1:SP))$
$(SP) \leftarrow (SP) + 2$

Flags: Affected— All

Encoding: 10011101
9D

Example: POPF

PUSH
Push Word onto Stack

Purpose: PUSH decrements the stack pointer SP by 2 and then transfers a word from the *source* operand to the stack element currently addressed by SP.

Format: PUSH *source*

Remarks: The stack pointer (SP) is decremented by 2. The contents of the specified operand are placed on the top of the stack at the location pointed to by SP. The contents of SP are used as an offset to the stack's base address in register SS.

There are three separate types of PUSH instructions depending on the type of operand supplied.

Logic: $(SP) \leftarrow (SP) - 2$
$((SP+1):(SP)) \leftarrow (SRC)$

Flags: None

Register Operand (Word):

Encoding: 01010reg
50 + reg

Example:

```
PUSH   AX ;(01010000)
          ;(   50   )
PUSH   SI ;(01010110)
          ;(   56   )
```

PUSH
Push Word onto Stack

Segment Register:

Encoding: See example.

Example:

```
PUSH   SS ;(00010110)
          ;(   16   )
PUSH   ES ;(00000110)
          ;(   06   )
```

Note: PUSH CS *is* valid.

Memory-or-Register Operand:

Encoding: 11111111 mod110r/m
FF mod110r/m

Example:

```
PUSH   BETA            ;(11111111  00110110)
                       ;(   FF        36   )
PUSH   BETA[BX]        ;(11111111  10110111)
                       ;(   FF        B7   )
PUSH   BETA[BX][DI]    ;(11111111  10110001)
                       ;(   FF        B1   )
```

PUSH (80286)
Push Immediate onto Stack

Purpose: The PUSH immediate instruction decrements the stack pointer SP by 2 and then transfers the immediate data to the stack element currently addressed by SP.

Format: PUSH *immediate*

Remarks: The data can be either immediate byte or immediate word. Byte data will be sign extended to a word before it is pushed onto the stack, because all stack operations are done on word data.

The stack pointer (SP) is decremented by 2. The contents of the specified operand are placed on the top of the stack at the location pointed to by SP. The contents of SP are used as an offset to the stack's base address in register SS.

Logic: (SP) ← (SP)-2
((SP+1):(SP))←(SRC)

Flags: None

Encoding: 011010s0 data [data if s=0]
68 + s data [data if s=0]

Example:

```
PUSH  279DH ;(01101000 00100111 10011101)
            ;(   68       27       9D   )
PUSH    9DH ;(01101010 10011101)
            ;(   6A       9D   )
```

PUSHA (80286)
Push All General Registers

Purpose: PUSHA saves the the contents of the eight general purpose registers AX, CX, DX, BX, original SP, BP, SI, and DI on the stack.

Format: PUSHA

Remarks: The stack pointer SP is decremented by 16 to hold the 8 word values. Since the registers are pushed onto the stack in the order above, they appear in the 16 new stack bytes in the reverse order.

> **Note:** PUSHA does not save any of the segment registers, the instruction pointer, or the flag register.

Logic: Push AX,CX,DX,BX,original SP,BP,SI,DI

Flags: None

Encoding: 01100000
60

Example:

```
PUSHA
```

PUSHF

Push Flags onto Stack

Purpose: PUSHF decrements the SP register by 2 and transfers all the flag registers into specific bits of the word operand (stack element) addressed by SP.

Format: PUSHF (no operands)

Remarks: The stack pointer is decremented by 2, then the flags replace the appropriate bits of the word at the top of the stack (see POPF).

Logic:
(SP) ← (SP) - 2
((SP)+1:(SP)) ← Flags

Flags: None

Encoding:
10011100
9C

Example: PUSHF

RCL
Rotate Left Through Carry

Purpose: RCL rotates the operand left through the CF flag register by COUNT bits.

Format:

RCL *destination*,1
or
RCL *destination*,CL

Remarks: The specified *destination* (leftmost) operand is rotated left through the carry flag a number of times (*count*). That number is either exactly 1, specified by an absolute number of value 1, or it is the number held in the CL register, specified by the right operand of CL.

CF is rotated into bit 0 of the *destination*. The highest-order bit of the *destination* is rotated into CF. All other bits in the *destination* move up one position (the value of the third bit is replaced by the value of the second bit). The rotation continues until the *count* is exhausted.

If the *count* was 1, and the two highest-order bits of the original *destination* value were unequal (one 0 and one 1), the OF is set to 0. If they were equal, OF is reset. If the *count* was not 1, OF is undefined and does not have a reliable value.

RCL
Rotate Left Through Carry

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← high-order bit of (EA)
  (EA) ← (EA)*2 + (tmpcf)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags: Affected— CF, OF

Encoding: 110100vw mod010r/m
D0 + vw mod010r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

RCL
Rotate Left Through Carry

Example: **Rotate register, count immediate:**

```
RCL      AH,1
RCL      BL,1
RCL      CX,1
VAL_ONE  EQU 1
RCL      DX,VAL_ONE
RCL      SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCL   MEM_BYTE,1
RCL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
RCL   DH,CL  ;ROTATES 3 BITS LEFT
RCL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL, 6
RCL   MEM_WORD,CL   ;ROTATES 6 TIMES
RCL   GAMMA_BYTE,CL
RCL   BETA[BX][DI],CL
```

RCL (80286)
Rotate Left Through Carry

Purpose:	RCL rotates the operand left through the CF flag register by COUNT bits.
Format:	RCL *destination*,1 or RCL *destination*,CL or RCL *destination,count*

Remarks: The specified *destination* (leftmost) operand is rotated left through the carry flag a number of times (*count*). That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL.

CF is rotated into bit 0 of the *destination*. The highest-order bit of the *destination* is rotated into CF. All other bits in the *destination* move up one position (the value of the third bit is replaced by the value of the second bit). The rotation continues until the *count* is exhausted.

If the *count* was 1, and the two highest-order bits of the original *destination* value were unequal (one 0 and one 1), the OF is set to 0. If they were equal, OF is reset. If the *count* was not 1, OF is undefined and does not have a reliable value.

> **Note:** The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

RCL (80286)
Rotate Left Through Carry

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← high-order bit of (EA)
  (EA) ← (EA)*2 + (tmpcf)
  (temp) ← (temp)-1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags: Affected— CF, OF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod010r/m
D0 + vw mod010r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod010r/m
C0 + w mod010r/m

RCL (80286)
Rotate Left Through Carry

Example: **Rotate register, count immediate:**

```
RCL      AH,1
RCL      BL,1
RCL      CX,1
RCL      CX,5
VAL_ONE  EQU 1
RCL      DX,VAL_ONE
RCL      SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCL   MEM_BYTE,1
RCL   MEM_BYTE,5
RCL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
RCL   DH,CL  ;ROTATES 3 BITS LEFT
RCL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL, 6
RCL   MEM_WORD,CL   ;ROTATES 6 TIMES
RCL   GAMMA_BYTE,CL
RCL   BETA[BX][DI],CL
```

RCR
Rotate Right Through Carry

Purpose: RCR rotates in EA operand right through the CF flag register by *count* bits.

Format: RCR *destination*,1
or
RCR *destination*,CL

Remarks: The specified *destination* (leftmost) operand is rotated right through the carry flag a number of times (*count*). That number is either exactly 1, specified by an absolute number of value 1, or it is the number held in the CL register, specified by a right operand of CL.

CF is preserved and is rotated into the high-order bit of the destination. The lowest-order bit of the destination is rotated into CF. All other bits in the *destination* move down one position (the value of the second bit is replaced by the value of the third bit). The rotation continues until the *count* is exhausted.

If the *count* was 1 and the two highest-order bits of the destination value are now unequal (one 0 and one 1), the overflow flag is set. If they were equal, OF is reset. If the *count* was not 1, OF is undefined and has no reliable value.

RCR
Rotate Right Through Carry

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2
  high-order bit of (EA) ← (tmpcf)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags: Affected— CF, OF

Encoding: 110100vw mod011r/m

D0 + vw mod011r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

RCR
Rotate Right Through Carry

Example: **Rotate register, count immediate:**

```
RCR      AH,1
RCR      BL,1
RCR      CX,1
VAL_ONE  EQU 1
RCR      DX,VAL_ONE
RCR      SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCR    MEM_BYTE,1
RCR    ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV    CL,3
RCR    DH,CL  ;ROTATES 3 BITS RIGHT
RCR    AX,CL
```

Rotate memory, count in CL:

```
MOV    CL, 6
RCR    MEM_WORD,CL   ;ROTATES 6 TIMES
RCR    GAMMA_BYTE,CL
RCR    BETA[BX][DI],CL
```

RCR (80286)
Rotate Right Through Carry

Purpose: RCR rotates in EA operand right through the CF flag register by *count* bits.

Format: RCR *destination*,1
or
RCR *destination*,CL
or
RCR *destination*,*count*

Remarks: The specified *destination* (leftmost) operand is rotated right through the carry flag a number of times (*count*). That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL.

The rotation continues until the *count* is exhausted. CF is preserved and is rotated into the high-order bit of the destination. The lowest-order bit of the destination is rotated into CF. All other bits in the *destination* move down one position (the value of the second bit is replaced by the value of the third bit). The rotation continues until the *count* is exhausted.

If the *count* was 1 and the two highest-order bits of the destination value are now unequal (one 0 and one 1), the overflow flag is set. If they were equal, OF is reset. If the *count* was not 1, OF is undefined and has no reliable value.

> **Note:** The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

RCR (80286)
Rotate Right Through Carry

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (tmpcf) ← (CF)
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2
  high-order bit of (EA) ← (tmpcf)
  (temp) ← (temp)-1
if COUNT = 1 then
  if high-order bit of (EA) ≠
       next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags: Affected— CF, OF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod011r/m
D0 + vw mod011r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod011r/m
C0 + w mod011r/m

RCR (80286)
Rotate Right Through Carry

Example: **Rotate register, count immediate:**

```
RCR      AH,1
RCR      BL,1
RCR      CX,1
RCR      CX,5
VAL_ONE  EQU 1
RCR      DX,VAL_ONE
RCR      SI,VAL_ONE
```

Rotate memory, count immediate:

```
RCR    MEM_BYTE,1
RCR    MEM_BYTE,5
RCR    ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV    CL,3
RCR    DH,CL  ;ROTATES 3 BITS RIGHT
RCR    AX,CL
```

Rotate memory, count in CL:

```
MOV    CL, 6
RCR    MEM_WORD,CL   ;ROTATES 6 TIMES
RCR    GAMMA_BYTE,CL
RCR    BETA[BX][DI],CL
```

REP/REPZ/REPE/REPNE/REPNZ
Repeat String Operation

Purpose: REP causes the primitive string operation that follows to be done repeatedly while (CX) is not zero. For CMPS and SCAS, if after any repetition of the primitive operation the ZF flag differs from the "z" bit of the repeat prefix, the repetition is ended. This prefix can be combined with the segment override and/or LOCK prefixes, although with multiple prefixes, interrupts must be disabled, because the return from an interrupt returns control to the interrupted instruction or to at most one prefix byte before that instruction.

Format:

REP (no operands) ;Set z bit to 1
or
REPZ (no operands) ;Set z bit to 1
or
REPE (no operands) ;Set z bit to 1
or
REPNE (no operands) ;Set z bit to 0
or
REPNZ (no operands) ;Set z bit to 0

Remarks: The specified string operation is carried out until (CX) is decremented to 0. CX is decremented by 1 after each iteration.

The compare and scan string operations exit the loop when the zero flag is unequal to the value of bit 0 of this instruction byte.

- If the count in CX ran out, CX will be zero, the index register will point one past the last byte of the string.

- If the ZF flag terminated the repeat, the index register will be one byte past the byte that caused the ZF condition, and CX will be correspondingly smaller, requiring further adjustments, for example:

```
        REPE    SCASB
        JE      X           ;EXIT BECAUSE UNEQUAL?
        DEC     DI          ;YES, ADJUST INDEX
        INC     CX          ;AND COUNT REG
X:
```

Logic:

```
Do while (CX) ≠ 0
   service pending interrupt (if any)
   execute primitive string operation
    in the succeeding byte
(CX) ← (CX) - 1
if primitive operation is CMPS, or
   SCAS and (ZF) ≠ z bit of the
   repeat prefix, then exit from while loop
```

Flags: See individual string operations.

Encoding: 1111001z
F2 + z

REP/REPZ/REPE/REPNE/REPNZ
Repeat String Operation

Example:

```
REP   MOVS  DEST,SOURCE  ;SEE MOVS

REPE  CMPS  DEST,SOURCE
;LOOP WILL BE EXITED BEFORE (CX)=0 ONLY
;IF (ZF)=0, ONLY IF THE BYTE AT (DI) IS NOT
;EQUAL TO THE BYTE AT (SI). SEE CMPS.

REPNZ  SCAS  DEST  ;SEE SCAS
;ONLY IF (ZF)=1, (AL) = DEST, WILL
; THIS LOOP BE EXITED BEFORE (CX) = 0
```

Note: REPNZ (nonzero) = REPNE (not equal)
REPZ (zero) = REPE (equal)

Code one of the forms of these instructions immediately preceding (but separated by at least one blank) the primitive string mnemonic (for example REPNZ SCASW). This specifies that the string operation is to be repeated the number of times determined by CX.

RET
Return from Procedure

Purpose: RET transfers control to the return address pushed by a previous CALL operation and optionally adds an immediate constant, *pop-value*, to the SP register to discard stack parameters. If this is an inter-segment RET, it was assembled under a procedure labeled FAR, it replaces the IP and the CS using the two words at the top of the stack. Otherwise, only the IP is replaced, using only one word from the top of the stack.

Format: RET [*pop-value*]

Remarks: The instruction pointer is replaced by the word at the top of the stack (offset of top is in the stack pointer). SP is incremented by 2. For inter-segment returns, the CS register is replaced by the word now at the top of the stack, and SP is again incremented by 2. If an immediate value *pop-value*, was specified on the RET statement, that value is now added to SP.

When using indirect CALLs, you must carefully ensure that the type of CALL matches the type of RETurn in the procedure:

```
CALL WORD PTR [BX]
```

must not invoke a FAR procedure. And;

```
CALL DWORD PTR [BX]
```

must not invoke a NEAR procedure.

RET
Return from Procedure

Logic:

```
(IP) ← ((SP)+1:(SP))
(SP) ← (SP) + 2
if inter-segment then
   (CS) ← ((SP)+1:(SP))
   (SP) ← (SP) + 2
if Add immediate to Stack Pointer
   then (SP) ← (SP) + data.
```

Flags: None.

Intra-Segment:

Encoding: 11000011
C3

Example:

```
RET
```

Intra-Segment and Add Immediate to Stack Pointer:

Encoding: 11000010 data-low data-high
C2 data-low data-high

Example:

```
RET   4
RET   12
```

> **Note:** These values cause 2 and 6 parameter words earlier stored on the stack to be discarded. Since most stack operations are on words, these values are usually even numbers (2 bytes per word).

RET
Return from Procedure

Inter-Segment and Add Immediate to Stack Pointer:

Encoding: 11001010 data-low data-high
CA data-low data-high

Example:

```
RET   2  ;INTER-SEGMENT RETURNS
RET   8  ;RESTORE IP FIRST, THEN CS
```

Inter-Segment:

Encoding: 11001011 (CBH)
CB (CBH)

Example:

```
RET
```

ROL
Rotate Left

Purpose: ROL rotates the operand left by *count* bits.

Format: ROL *destination,count*

Remarks: The specified destination (leftmost) operand is rotated left *count* times.

Its high-order bit replaces the carry flag, whose original value is lost.

All other bits in the destination move up one position (the value of the third bit is replaced by the value of the second bit). The vacated bit-position 0 is filled by the new CF (the old high-order bit).

If *count* is greater than 1, set *count* into CL before issuing ROL; then specify CL as the second operand. If *count* is 1, the second operand is a 1.

The rotation continues until the *count* is exhausted. If *count* was 1 and the new value of CF is not equal to the new high-order bit, the overflow flag is set; if (CF) does equal that high-order bit, OF becomes 0. However, if *count* was not 1, OF is not defined and has no reliable value.

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA) * 2 + (CF)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

ROL
Rotate Left

Flags: Affected— CF, OF

Encoding: 110100vw mod000r/m
D0 + vw mod000r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example: **Rotate register, count immediate:**

```
ROL        AH,1
ROL        BL,1
ROL        CX,1
VAL_ONE    EQU  1
ROL        DX,VAL_ONE
ROL        DI,VAL_ONE
```

Rotate memory, count immediate:

```
ROL   MEM_BYTE,1
ROL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROL   DH,CL  ;ROTATES 3 BITS LEFT
ROL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROL   MEM_WORD,CL    ;ROTATES 6 TIMES
ROL   GAMMA_BYTE,CL
ROL   BETA[BX][DI],CL
```

ROL (80286)
Rotate Left

Purpose: The specified destination (leftmost) operand is rotated left *count* times.

Format:

ROL *destination*,1
or
ROL *destination*,CL
or
ROL *destination,count*

Remarks: That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL.

If *count* is 1, the second operand is a 1, then the destination's high-order bit replaces the carry flag, whose original value is lost. All other bits in the destination move up one position (the value of the third bit is replaced by the value of the second bit). The vacated bit-position 0 is filled by the new CF (the old high-order bit).

The rotation continues until the *count* is exhausted. If *count* was 1 and the new value of CF is not equal to the new high-order bit, the overflow flag is set; if (CF) does equal that high-order bit, OF becomes 0. However, if *count* was not 1, OF is not defined and has no reliable value.

> **Note:** The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

ROL (80286)
Rotate Left

Logic:

```
(temp) <- COUNT
do while (temp) ≠ 0
  (CF) <- high-order bit of (EA)
  (EA) <- (EA) * 2 + (CF)
  (temp) <- (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) <- 1
  else (OF) <- 0
else (OF) undefined
```

Flags: Affected— CF, OF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod000r/m
D0 + vw mod000r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod000r/m
C0 + w mod000r/m

Example: **Rotate register, count immediate:**

```
ROL       AH,1
ROL       BL,1
ROL       CX,1
ROL       CX,5
VAL_ONE   EQU  1
ROL       DX,VAL_ONE
ROL       DI,VAL_ONE
```

ROL (80286)
Rotate Left

Rotate memory, count immediate:

```
ROL   MEM_BYTE,1
ROL   MEM_BYTE,5
ROL   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROL   DH,CL  ;ROTATES 3 BITS LEFT
ROL   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROL   MEM_WORD,CL   ;ROTATES 6 TIMES
ROL   GAMMA_BYTE,CL
ROL   BETA[BX][DI],CL
```

ROR
Rotate Right

Purpose: ROR rotates the operand right by *count* bits.

Format: ROR *destination,count*

Remarks: The specified destination (leftmost) operand is rotated right *count* times. Its low-order bit replaces the carry flag, whose original value is lost. All other bits in the destination move down one position, (the value of the second bit is replaced by the value of the third bit). The vacated high-order position is filled by the new CF (the old value of position 0).

If *count* is greater than 1, set *count* into CL before issuing ROR; then specify CL as the second operand. If *count* is 1, the second operand is a 1.

The rotation continues until the *count* is exhausted. If *count* was 1 and the new high-order value is not equal to the old high-order value, the overflow flag is set; if they are equal, (OF) = 0. However, if *count* was not 1, OF is undefined and has no reliable value.

Logic:

```
(temp) ← COUNT
DO WHILE (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2
  high-order bit of (EA) ← (CF)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

ROR
Rotate Right

Flags: Affected— CF, OF

Encoding: 110100vw mod001r/m
D0 + vw mod001r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example: **Rotate register, count immediate:**

```
ROR       AH,1
ROR       BL,1
ROR       CX,1
VAL_ONE   EQU  1
ROR       DX,VAL_ONE
ROR       SI,VAL_ONE
```

Rotate memory, count immediate:

```
ROR   MEM_BYTE,1
ROR   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROR   DH,CL  ;ROTATES 3 BITS RIGHT
ROR   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROR   MEM_WORD,CL   ;ROTATES 6 TIMES
ROR   GAMMA_BYTE,CL
ROR   BETA[BX][DI],CL
```

ROR (80286)
Rotate Right

Purpose: The specified destination (leftmost) operand is rotated right *count* times.

Format:
ROR *destination*,1
or
ROR *destination*,CL
or
ROR *destination,count*

Remarks: That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL.

If *count* is 1, the second operand is a 1, then the destination's low-order bit replaces the carry flag, whose original value is lost. All other bits in the destination move down one position, (the value of the second bit is replaced by the value of the third bit). The vacated high-order position is filled by the new CF (the old value of position 0).

The rotation continues until the *count* is exhausted. If *count* was 1 and the new high-order value is not equal to the old high-order value, the overflow flag is set; if they are equal, (OF) ← 0. However, if *count* was not 1, OF is undefined and has no reliable value.

> **Note:** The 80286 does not allow rotation counts greater than 31. Only the lower 5 bits of the rotation count are used if a rotation count greater than 31 is attempted. The 8088 does not mask rotation counts.

ROR (80286)
Rotate Right

Logic:

```
(temp) ← COUNT
DO WHILE (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2
  high-order bit of (EA) ← (CF)
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags: Affected— CF, OF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod001r/m
D0 + vw mod001r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod001r/m
C0 + w mod001r/m

Example: **Rotate register, count immediate:**

```
ROR        AH,1
ROR        BL,1
ROR        CX,1
ROR        CX,5
VAL_ONE    EQU  1
ROR        DX,VAL_ONE
ROR        SI,VAL_ONE
```

ROR (80286)
Rotate Right

Rotate memory, count immediate:

```
ROR   MEM_BYTE,1
ROR   MEM_BYTE,5
ROR   ALPHA[DI],VAL_ONE
```

Rotate register, count in CL:

```
MOV   CL,3
ROR   DH,CL  ;ROTATES 3 BITS RIGHT
ROR   AX,CL
```

Rotate memory, count in CL:

```
MOV   CL,6
ROR   MEM_WORD,CL   ;ROTATES 6 TIMES
ROR   GAMMA_BYTE,CL
ROR   BETA[BX][DI],CL
```

SAHF
Store AH in Flags

Purpose: SAHF transfers specific bits of the AH register to the flag registers SF, ZF, AF, PF, and CF. The bits of AH indicated by "X" in the operation are ignored.

Format: SAHF (no operands)

Remarks: The five flags shown are replaced by specified bits from AH, the high-order byte of the accumulator:

```
(SF) = bit 7
(ZF) = bit 6
(AF) = bit 4
(PF) = bit 2
(CF) = bit 0
```

Logic: (SF):(ZF):X:(AF):X:(PF):X:(CF)←(AH)

Flags: Affected— AF, CF, PF, SF, ZF

Encoding: 10011110
9E

Example:

```
SAHF
```

SAL/SHL
Shift Arithmetic Left/Logical Left

Purpose: SAL and SHL shift the operand left by *count* bits; shifting in low-order zero bits.

Format: SAL *destination,count*
or
SHL *destination,count*

Remarks: The specified *destination* (leftmost) operand is shifted left *count* times. Its high-order bit replaces the carry flag, whose original value is lost. All other bits in the destination move up one position (the value of the third bit is replaced by the value of the second bit). The vacated low-order bit-position is filled by 0. *count* can be an immediate 1, or the value can be contained in CL. (See example.)

If *count* is greater than 1, set *count* into CL before issuing SAL (or SHL); then specify CL as the second operand. If *count* is 1, the second operand is a 1.

The shift continues until the *count* is exhausted. If *count* was 1 and the new value of CF is not equal to the new high-order bit, the overflow flag is set; if (CF) does not equal that high-order bit, OF becomes 0. However, if *count* was not 1, OF is not defined and has no reliable value.

> **Note:** If *count* is 1, code it as immediate; if *count* is greater than 1, move *count* to CL and code CL in the second operand position (*count*)

SAL/SHL
Shift Arithmetic Left/Logical Left

Logic:

```
(temp) ← (COUNT)
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA)*2
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags:

Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding:

110100vw mod100r/m
D0 + vw mod100r/m
If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Example: **Shift register, count immediate:**

```
SHL       AH,1
SHL       BL,1
SHL       CX,1
VAL_ONE   EQU  1
SHL       SI,VAL_ONE
SHL       SI,VAL_ONE
```

SAL/SHL
Shift Arithmetic Left/Logical Left

Shift memory, count immediate:

```
SHL  MEM_BYTE,1
SHL  ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV  CL,3
SHL  DH,CL ;SHIFT 3 BITS LEFT
SHL  AX,CL
```

Shift register, count in CL:

```
MOV  CL,6
SHL  MEM_WORD,CL ;SHIFT 6 TIMES
SHL  GAMMA_BYTE,CL
SHL  BETA[BX][DI],CL
```

SAL/SHL (80286)
Shift Arithmetic Left/Shift Logical Left

Purpose: SAL and SHL shift the operand left by *count* bits; shifting in low-order zero bits.

Format:

SAL *destination*,1
 or
SAL *destination*,CL
 or
SAL *destination,count*

SHL *destination*,1
 or
SHL *destination*,CL
 or
SHL *destination,count*

Remarks: The specified *destination* (leftmost) operand is shifted left *count* times.

If *count* is 1, the second operand is a 1, then the destination's high-order bit replaces the carry flag, whose original value is lost. All other bits in the destination move up one position (the value of the third bit is replaced by the value of the second bit). The vacated low-order bit-position is filled by 0.

That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL. (See example.)

The shift continues until the *count* is exhausted. If *count* was 1 and the new value of CF is not equal to the new high-order bit, the overflow flag is set; if (CF) does not equal that high-order bit, OF becomes 0. However, if *count* was not 1, OF is not defined and has no reliable value.

SAL/SHL (80286)
Shift Arithmetic Left/Shift Logical Left

If *count* is greater than 31, set *count* into CL before issuing SHR; then specify CL as the second operand. If count is 1-31, the second operand can be 1-31

> **Note:** The 80286 does not allow shift counts greater than 31. Only the lower 5 bits of the shift count are used if a shift count greater than 31 is attempted. The 8088 uses all 8 bits of the shift count.

Logic:

```
(temp) ← (COUNT)
do while (temp) ≠ 0
  (CF) ← high-order bit of (EA)
  (EA) ← (EA)*2
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠ (CF)
    then (OF) ← 1
  else (OF) ← 0
else (OF) undefined
```

Flags: Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod100r/m
D0 + vw mod100r/m
If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod100r/m
C0 + w mod100r/m

SAL/SHL (80286)

Shift Arithmetic Left/Shift Logical Left

Example: **Shift register, count immediate:**

```
SHL      AH,1
SHL      BL,1
SHL      CX,1
SHL      CX,5
VAL_ONE  EQU  1
SHL      SI,VAL_ONE
SHL      SI,VAL_ONE
```

Shift memory, count immediate:

```
SHL  MEM_BYTE,1
SHL  MEM_BYTE,5
SHL  ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV  CL,3
SHL  DH,CL ;SHIFT 3 BITS LEFT
SHL  AX,CL
```

Shift register, count in CL:

```
MOV  CL,6
SHL  MEM_WORD,CL ;SHIFT 6 TIMES
SHL  GAMMA_BYTE,CL
SHL  BETA[BX][DI],CL
```

SAR
Shift Arithmetic Right

Purpose: SAR (shift arithmetic right) shifts the *destination* operand right by *count* bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

Format: SAR *destination,count*

Remarks: The specified *destination* (leftmost) operand is shifted right *count* times. Its low-order bit replaces the carry flag, whose original value is lost. All other bits in the *destination* move down one position (the value of the second bit is replaced by the value of the third bit). The vacated high-order position retains its old value. (If the original high-order bit value was 0, zeroes are shifted in). If that value was 1, 1's are shifted in.

If *count* is greater than 1, set *count* into CL before issuing SAR; then specify CL as the second operand. If *count* is 1, the second operand is a 1 (see the following example.)

The shift continues until the *count* is exhausted. If *count* was 1 and the high-order value is not equal to the next-to-high-order value, the overflow flag is set; if they are equal, (OF) = 0. However, if *count* was not 1, OF is reset.

Flags: Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding: 110100vw mod111r/m
D0 + vw mod111r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

SAR
Shift Arithmetic Right

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
(CF) ← low-order bit of (EA)
(EA) ← (EA)/2, where / is equivalent to signed
   division, rounding down
(temp) ← (temp) - 1
if COUNT = 1 then
if high-order bit of (EA) ≠
     next-to-high-order bit of (EA)
   then (OF) ← 1
   else (OF) ← 0
else (OF) ← 0
```

Example: **Shift register, count immediate:**

```
SAR       AH,1
SAR       BL,1
SAR       CX,1
VAL_ONE   EQU 1
SAR       DX,VAL_ONE
SAR       SI,VAL_ONE
```

Shift memory, count immediate:

```
SAR   MEM_BYTE,1
SAR   ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SAR   DH,CL  ;SHIFTS 3 BITS RIGHT
SAR   AX,CL
```

SAR (80286)
Shift Arithmetic Right

Purpose: SAR (shift arithmetic right) shifts the *destination* operand right by *count* bits, shifting in high-order bits equal to the original high-order bit of the operand (sign extension).

Format:

SAR *destination*,1
or
SAR *destination*,CL
or
SAR *destination,count*

Remarks: The specified *destination* (leftmost) operand is shifted right *count* times. That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL.

If *count* is 1, the second operand is a 1, then the destination's low-order bit replaces the carry flag, whose original value is lost. All other bits in the *destination* move down one position (the value of the second bit is replaced by the value of the third bit). The vacated high-order position retains its old value. (If the original high-order bit value was 0, zeroes are shifted in). If that value was 1, 1's are shifted in.

The shift continues until the *count* is exhausted. If *count* was 1 and the high-order value is not equal to the next-to-high-order value, the overflow flag is set; if they are equal, (OF) ← 0. However, if *count* was not 1, OF is reset.

If *count* is greater than 31, set *count* into CL before issuing SHR; then specify CL as the second operand. If count is 1-31, the second operand can be 1-31

SAR (80286)
Shift Arithmetic Right

Note: The 80286 does not allow shift counts greater than 31. Only the lower 5 bits of the shift count are used if a shift count greater than 31 is attempted. The 8088 uses all 8 bits of the shift count.

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2, where / is equivalent to signed
    division, rounding down
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) ← 0
```

Flags: Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod111r/m
D0 + vw mod111r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod111r/m
C0 + w mod111r/m

SAR (80286)
Shift Arithmetic Right

Example: **Shift register, count immediate:**

```
SAR      AH,1
SAR      BL,1
SAR      CX,1
SAR      CX,5
VAL_ONE  EQU 1
SAR      DX,VAL_ONE
SAR      SI,VAL_ONE
```

Shift memory, count immediate:

```
SAR   MEM_BYTE,1
SAR   MEM_BYTE,5
SAR   ALPHA[DI],VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SAR   DH,CL  ;SHIFTS 3 BITS RIGHT
SAR   AX,CL
```

SBB
Subtract with Borrow

Purpose: SBB does a subtraction of the two operands, subtracts one if the CF flag is set, and returns the result to one of the operands.

Format: SBB *destination,source*

Remarks: The *source* (rightmost) operand is subtracted from the *destination* (leftmost). If the carry flag was set, 1 is subtracted from the above result. The result replaces the original *destination* operand.

Logic: If (CF) = 1, (DEST) ← (LSRC) - (RSRC) - 1
If (CF) = 0, (DEST) ← (LSRC) - (RSRC).

Flags: Affected—AF, CF, OF, PF, SF, ZF

Memory or Register Operand and Register Operand:

Encoding: 000110dw modregr/m
18 + dw modregr/m

If d = 1, LSRC = REG, RSRC = EA, DEST = REG.
If d = 0, LSRC = EA, RSRC = REG, DEST = EA.

Example: **Subtract register from register:**

```
SBB   AX,BX
SBB   CH,DL
```

Subtract memory from register:

```
SBB   DX,MEM_WORD
SBB   DI,ALPHA[SI]
SBB   DL,MEM_BYTE[DI]
```

Subtract register from memory:

```
SBB   MEM_WORD,AX
SBB   MEM_BYTE[DI],BX
SBB   GAMMA[BX][DI],SI
```

Immediate Operand from Accumulator:

Encoding: 0001110w data data*
1C + w data data*

If w = 0, then LSRC = data, DEST = AL.
If w = 1, then LSRC = data, DEST = AX.

*Present only if w=1.

Example: **Subtract immediate (byte):**

```
SBB         AL,4
VAL_SIXTY   EQU 60
SBB         AL,VAL_SIXTY
```

Subtract immediate (word):

```
SBB   AX,660
SBB   AX,VAL_SIXTY*6
```

SBB
Subtract with Borrow

Immediate Operand from Memory or Register Operand:

Encoding: 100000sw mod011r/m data
80 + sw mod011r/m data

LSRC = EA, RSRC = data, DEST = EA

If an immediate-data-byte is being subtracted from a register-or-memory word, the byte is sign-extended to 16 bits before the subtraction. For this situation, the instruction byte is 83H (the s and w bits are both set).

Example: **Subtract immediate from register:**

```
SBB   BX,2001
SBB   CL,VAL_SIXTY
SBB   SI,VAL_SIXTY*9
```

Subtract immediate from memory:

```
SBB   MEM_BYTE,12
SBB   MEM_BYTE[DI],VAL_SIXTY
SBB   MEM_WORD[BX],79
SBB   GAMMA[DI][BX],1984
```

SCAS/SCASB/SCASW
Scan Byte or Word String

Purpose: SCAS subtracts the *destination* byte (or word) operand addressed by DI from AL (or AX) and affects the flags but does not return the result. As a repeated operation this provides for scanning for the occurrence of, or departure from, a given value in a string. See "REP" instruction in this chapter.

Format:

SCAS *dest-string*
  or
SCASB (no operands)
  or
SCASW (no operands)

Remarks: The *destination-string* element specified by DI in the extra segment is subtracted from the value in the accumulator but the operation affects flags only. The destination index is then incremented (if the direction flag is zero) or decremented (if (DF) = 1) by 1 for byte strings or 2 for words. (See the CLD and STD instructions.)

Logic:

```
(LSRC) - (RSRC)
if (DF) = 0, then
  (DI) ← (DI) + DELTA
else (DI) ← (DI) - DELTA
```

Flags: Affected— AF, CF, OF, PF, SF, ZF

SCAS/SCASB/SCASW
Scan Byte or Word String

Encoding: 1010111w
AE + w

If w=0,
LSRC = AL, RSRC = (DI), DELTA = 1.
If w=1,
LRSC = AX, RSRC = (D) + 1:(DI), DELTA = 2.

Example:

```
CLD   ;CLEARS DF, CAUSES DI INCREMENTING
MOV   DI,OFFSET DEST_BYTE_STRING
MOV   AL,'M'
SCAS  DEST_BYTE_STRING
    or
SCAS  ES:BYTE PTR[DI]
    or
SCASB

STD   ;SETS DF, CAUSES DI DECREMENTING
MOV   DI,OFFSET WORD_STRING
MOV   AX,'MD'
SCAS WORD_STRING
```

The operand named in the SCAS instruction is used only by the assembler to verify type and accessibility using current segment register contents. The actual operation of this instruction uses DI to point to the location to be scanned, without using the operand named in the source line.

The string instructions are unusual in several aspects:

1. Load SI with the offset of the *source-string*.

2. Load DI with the offset of the *destination-string*.

3. Each can be coded with or without symbolic memory operand.

   - If symbolic operand is coded, the assembler can check the addressability of it for you.

   - References that use hardware defaults should be coded using the operand-less forms (SCASB and SCASW), to avoid the additional pointer information.

   - Never use [BX], [SI], or [BP] addressing modes with the string instructions.

4. If the instruction mnemonic is coded without operands, the segment registers are as follows:

   - SI defaults to an offset in the segment addressed by DS.

   - DI is required to be an offset in the segment addressed by ES.

SHR
Shift Logical Right

Purpose: SHR shifts the operand right shifting in high-order zero bits.

Format: SHR *destination,count*

Remarks: The specified *destination* (leftmost) operand is shifted right *count* times. Its low-order bit replaces the carry flag, whose original value is lost. All other bits in the *destination* move down one position. For example, the value of the second bit is replaced by the value of the third bit. The vacated high-order position is filled by 0.

The shift continues until the *count* is exhausted. If *count* was 1 and the new high-order value is not equal to the next-to-high-order value, then the overflow flag is set; if they are equal, (OF) = 0. However, if *count* was not 1, then OF is undefined and has no reliable value.

If *count* is greater than 1, set *count* into CL before issuing SHR; then specify CL as the second operand. If count is 1, the second operand can be a 1 (see example)..)

Flags: Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding:

110100vw	mod101r/m
D0 + vw	mod101r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

SHR
Shift Logical Right

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2, where / is equivalent
    to unsigned division
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) ← 1
   else (OF) ← 0
else (OF) ← 0
```

Example: **Shift register, count immediate:**

```
SHR      AH,1
SHR      BL,1
SHR      CX,1
VAL_ONE  EQU 1
SHR      DX,VAL_ONE
SHR      SI,VAL_ONE
```

Shift memory, count immediate:

```
SHR   MEM_BYTE,1
SHR   ALPHA[DI[,VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SHR   DH,CL  ;SHIFTS 3 BITS RIGHT
SHR   AX,CL
```

SHR (80286)
Shift Logical Right

Purpose: SHR shifts the operand right shifting in high-order zero bits.

Format: SHR *destination*,1
or
SHR *destination*,CL
or
SHR *destination,count*

Remarks: The specified *destination* (leftmost) operand is shifted right *count* times. That *count* can be an absolute number with a value from 1 to 31, specified as an immediate value, or it is the number held in the CL register, specified by the right operand of CL.

If *count* is 1, the second operand is a 1, then the destination's low-order bit replaces the carry flag, whose original value is lost. All other bits in the *destination* move down one position. For example, the value of the second bit is replaced by the value of the third bit. The vacated high-order position is filled by 0.

The shift continues until the *count* is exhausted. If *count* was 1 and the new high-order value is not equal to the next-to-high-order value, then the overflow flag is set; if they are equal, (OF) ← 0. However, if *count* was not 1, then OF is undefined and has no reliable value.

If *count* is greater than 31, set *count* into CL before issuing SHR; then specify CL as the second operand. If count is 1-31, the second operand can be 1-31 (see example)..)

SHR (80286)
Shift Logical Right

Note: The 80286 does not allow shift counts greater than 31. Only the lower 5 bits of the shift count are used if a shift count greater than 31 is attempted. The 8088 uses all 8 bits of the shift count.

Logic:

```
(temp) ← COUNT
do while (temp) ≠ 0
  (CF) ← low-order bit of (EA)
  (EA) ← (EA)/2, where / is equivalent to signed
    division, rounding down
  (temp) ← (temp) - 1
if COUNT = 1 then
  if high-order bit of (EA) ≠
      next-to-high-order bit of (EA)
    then (OF) ← 1
  else (OF) ← 0
else (OF) ← 0
```

Flags: Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding: For an immediate value of 1, or *count* of CL:
110100vw mod101r/m
D0 + vw mod101r/m

If v = 0, COUNT = 1.
If v = 1, COUNT = (CL).

Encoding: For an immediate value of 2-31:
1100000w mod101r/m
C0 + w mod101r/m

SHR (80286)
Shift Logical Right

Example: **Shift register, count immediate:**

```
SHR      AH,1
SHR      BL,1
SHR      CX,1
SHR      CX,5
VAL_ONE  EQU 1
SHR      DX,VAL_ONE
SHR      SI,VAL_ONE
```

Shift memory, count immediate:

```
SHR   MEM_BYTE,1
SHR   MEM_BYTE,5
SHR   ALPHA[DI[,VAL_ONE
```

Shift register, count in CL:

```
MOV   CL,3
SHR   DH,CL  ;SHIFTS 3 BITS RIGHT
SHR   AX,CL
```

STC
Set Carry Flag

Purpose: STC sets the CF flag.

Format: STC (no operand)

Remarks: The carry flag is set to 1.

Note: See CLC for the opposite function.

Logic: (CF) ← 1

Flags: Affected— CF

Encoding: 11111001
F9

Example:

```
STC
```

STD
Set Direction Flag

Purpose: STD sets the DF flag causing the string operations to auto-decrement the operand indexes.

Format: STD (no operands)

Remarks: The direction flag is set to 1.

Note: See CLD for the opposite function.

Logic: (DF) ← 1

Flags: Affected— DF

Encoding: 11111101
FD

Example:

```
STD   ;CAUSES DECREMENTING OF DI (AND SI)
;      IN STRING OPERATIONS
```

STI
Set Interrupt Flag (Enable)

Purpose:	STI sets the IF flag, enabling maskable external interrupts after the execution of the next instruction.
Format:	STI (no operands)
Remarks:	The interrupt flag is set to 1. Note: See CLI for the opposite function.
Logic:	(IF) ← 1
Flags:	Affected— IF
Encoding:	11111011 FB
Example:	`STI ;ENABLES INTERRUPTS`

STOS/STOSB/STOSW
Store Byte or Word String

Purpose: STOS transfers a byte (or word) operand from AL (or AX) to the *destination* operand addressed by DI and adjusts the DI register by DELTA. As a repeated operation (see "REP") this provides for filling a string with a given value. The operand named in the STOS instruction is used only by the assembler to verify type and accessibility using current segment register contents. The actual operation of the instruction uses only DI to point to the location being stored into.

Format:

STOS *destination-string*
  or
STOSB (no operands)
  or
STOSW (no operands)

Remarks: The byte (or word) in AL (or AX) replaces the contents of the byte (or word) pointed to by DI in the extra segment. DI is then incremented if the direction flag is zero or decremented if DF = 1. The change is 1 for bytes, 2 for words.

Logic:

```
(DEST) ← (SRC)
if (DF) = 0 then
  (DI) ← (DI) + DELTA
else (DI) ← (DI) - DELTA
```

Flags: None

STOS/STOSB/STOSW

Store Byte or Word String

Encoding: 1010101w
AA + w

If w=0, SRC = AL, DEST = 1.
If w=1, SRC = AX,
DEST = (DI) + 1:(DI), DELTA = 2.

Example: Store byte:

```
MOV   DI,OFFSET  BYTE_DEST_STRING
STOS  BYTE_DEST_STRING
```

Store word:

```
MOV   DI,OFFSET WORD_DEST
STOS  WORD_DEST
```

The string instructions are unusual in several aspects:

1. Load SI with the offset of the *source-string*.

2. Load DI with the offset of the *destination-string*.

3. Each can be coded with or without symbolic memory operands.

    - If symbolic operands are coded, the assembler can check the addressability of them for you.

    - References that use hardware defaults should be coded using the operand-less forms (STOSB and STOSW), to avoid the additional pointer information.

    - Never use [BX] or [BP] addressing modes with the string instructions.

4. If the instruction mnemonic is coded without operands, the segment registers are as follows:
   - SI defaults to an offset in the segment addressed by DS.
   - DI is required to be an offset in the segment addressed by ES.

SUB
Subtract

Purpose: SUB does a subtraction of the *source* operand from the *destination*. The result goes to the *destination* operand.

Format: SUB *destination,source*

Remarks: The *source* (rightmost) operand is subtracted from the *destination* (leftmost) operand and the result is stored in the *destination*.

Logic: (DEST) ← (LSRC) - (RSRC)

Flags: Affected: AF, CF, OF, PF, SF, ZF

Memory or Register Operand and Register Operand:

Encoding: 001010dw modregr/m
28 + dw modregr/m

If d = 1,
LSRC = REG, RSRC = EA, DEST = REG.
If d = 0,
LSRC = EA, RSRC = REG, DEST = EA.

Example: **Subtract register from register:**

```
SUB   AX,BX
SUB   CH,DL
```

Subtract memory from register:

```
SUB   DX,MEM_WORD
SUB   DI,ALPHA[SI]
SUB   BL,MEM_BYTE[DI]
```

SUB
Subtract

Subtract register from memory:

```
SUB   MEM_WORD,AX
SUB   MEM_BYTE[DI],BL
SUB   GAMMA[BX][DI],SI
```

Immediate Operand from Accumulator:

Encoding: 0010110w data
2C + w data

If w = 0, LSRC = AL, RSRC = data, DEST = AL.
If w = 1, LSRC = AX, RSRC = data, DEST = AX.

Example: **Subtract immediate from register (byte):**

```
SUB         AL,4
VAL_SIXTY   EQU 60
SUB         AL,VAL_SIXTY
```

Subtract immediate from register (word):

```
SUB   AX,660
SUB   AX,VAL_SIXTY*6
SUB   AX,6606
```

If an immediate-data-byte is being subtracted from a register-or-memory word, the byte is signed-extended to 16 bits before the subtraction. For this situation, the instruction is 83H (the s and w bits are both set).

SUB
Subtract

Immediate Operand from Memory or Register Operand:

Encoding: 100000sw mod101r/m data
80 + sw mod101r/m data

LSRC = EA, RSRC = data, DEST = EA

Example: **Subtract immediate from register:**

```
SUB   BX,2001
SUBM  CL,VAL_SIXTY
SUB   SI,VAL_SIXTY*9
```

Subtract immediate from memory:

```
SUB   MEM_BYTE,12
SUB   MEM_BYTE[DI],VAL_SIXTY
SUB   MEM_WORD[BX],79
SUB   GAMMA[DI][BX],1984
```

TEST
Test (Logical Compare)

Purpose: TEST does the bit-to-bit logical conjunction of the two operands, causing the flags to be affected, but does not return the result.

Format: TEST *destination,source*

Remarks: The operands are ANDed to affect the flags but neither operand is changed. The carry and overflow flags are reset.

The *source* (rightmost) operand must be of the same type (byte or word) as the *destination* operand. The only exception for TEST is testing an immediate-data byte with a memory word.

Logic: (LSRC) & (RSRC)
(CF) ← 0
(OF) ← 0

Flags: Affected— CF, OF, PF, SF, ZF
Undefined— AF

Memory or Register Operand with Register Operand:

Encoding: 1000010w modregr/m
84 + w modregr/m

LSRC = REG, RSRC = EA

TEST
Test (Logical Compare)

Example: **Register with register:**

```
TEST   AX,DX
TEST   SI,BP
TEST   BH,CL
```

Register with memory:

```
TEST   MEM_WORD,SI
TEST   MEM_BYTE,CH
TEST   ALPHA[DI],DX
TEST   BETA[BX][SI],CX
```

Memory with register:

```
TEST   DI,MEM_WORD
TEST   CH,MEM_BYTE
TEST   AX,GAMMA[BP][SI]
```

Immediate Operand with Accumulator:

Encoding: 1010100w data

A8 + w data

If w = 0, LSRC = AL, RSRC = data.
If w = 1, LSRC = AX, RSRC = data.

Example:

```
TEST   AL,6
TEST   AL,IMM_VALUE_DRIVE
TEST   AX,999
```

TEST
Test (Logical Compare)

Immediate Operand with Memory or Register Operand:

Encoding: 1111011w mod000r/m data
F6 + w mod000r/m data

LSRC = EA, RSRC = data

Example: Immediate with register:

```
TEST   BH,7
TEST   CL,19_IMM_BYTE
TEST   DX,IMM_DATA_WORD
TEST   SI,798
```

Immediate with memory:

```
TEST   MEM_WORD,IMM_DATA_BYTE
TEST   GAMMA[BX],IMM_BYTE
TEST   [BP][DI],6ACEH
```

WAIT
Wait

Purpose: This instruction allows the processor to synchronize itself with external hardware by placing the processor in a wait state until an external interrupt occurs.

Format: WAIT (no operands)

Flags: None

Encoding: 10011011
9B

Example:

```
WAIT
```

XCHG
Exchange

Purpose: XCHG exchanges the byte or word *source* operand with the *destination* operand.

Format: XCHG *destination,source*

Remarks: The segment registers cannot be operands of XCHG.

There are two forms of the XCHG instruction, one for switching the contents of the accumulator with those of some other general word register, and one for switching a register and a memory-or-register operand.

The exchange is performed by the following operations:

1. The contents of the *destination* (leftmost operand) are temporarily stored in an internal word register:

   `(Temp) ← (DEST)`

2. The contents of the *destination* are replaced by the contents of the *source* (rightmost) operand:

   `(DEST) ← (SRC)`

3. The former contents of the *destination* are moved from the work register into the *source* operand:

   `(SRC) ← (Temp)`

XCHG
Exchange

Logic: (Temp) ← (DEST)
(DEST) ← (SRC)
(SRC) ← (Temp)

Flags: None.

Register Operand with Accumulator:

Encoding: 10010reg
90 + reg

```
XCHG   AX,BX
XCHG   SI,AX
XCHG   CX,AX
```

Memory or Register Operand with Register Operand:

Encoding: 1000011w modregr/m
86 + w modregr/m

SRC = EA, DEST = REG

Example:

```
XCHG   BETA_WORD,CX
XCHG   BX,DELTA_WORD
XCHG   DH,ALPHA_BYTE
XCHG   BL,AL
```

XLAT
Translate

Purpose: XLAT does a table lookup byte translation. The AL register is used as an index into a table (256 bytes at most) addressed by the BX register. The byte operand so addressed is transferred to AL.

Format: XLAT *source-table*

Remarks: The operand is the variable name whose address was moved to BX. The SEG component of this operand lets the assembler determine if the table's segment is currently accessible. The TYPE attribute of the operand must be BYTE, or explicitly overridden with BYTE PTR. The contents of the accumulator are replaced by a byte from a table. The table's starting address has been moved into register BX. The original contents of AL is the number of bytes past that starting address, where the desired translation byte is to be found. It replaces the contents of AL.

Logic: (AL) ← ((BX) + (AL))

Flags: None.

Encoding: 11010111
D7

Example:

```
MOV   AL,IMMED_BYTE
MOV   BX, OFFSET TABLE_NAME
XLAT  TABLE_NAME
```

XOR
Exclusive OR

Purpose: XOR does the bit-to-bit logical exclusive dis-junction of the operands and returns the result to the *destination* operand.

Format: XOR *destination,source*

Remarks: Each bit position in the *destination* (leftmost) operand is set to zero if the corresponding bit positions in both operands are equal. If they are unequal, the bit position is set to 1.

Logic:
(DEST) ← (LSRC) XOR (RSRC)
(CF) ← 0
(OF) ← 0

Flags:
Affected— CF, OF, PF, SF, ZF
Undefined— AF

Encoding:
001100dw modregr/m
30 + dw modregr/m

If d = 1, LSRC = REG, RSRC = EA, DEST = REG.
If d = 0, LSRC = EA, RSRC = REG, DEST = EA.

Example: **Register with register:**

```
XOR   AH,BL  ;RESULT IN AH, BL UNCHANGED
XOR   SI,DX  ;RESULT IN SI, DX UNCHANGED
XOR   CX,DI  ;RESULT IN CX, DI UNCHANGED
```

Memory with register:

```
XOR   AX,MEM_WORD
XOR   CL,MEM_BYTE[SI]
XOR   SI,ALPHA[BX][SI]
```

XOR
Exclusive OR

Register with memory:

```
XOR    BETA[BX][DI],AX
XOR    MEM_BYTE,DH
XOR    GAMMA[DI],BX
```

Immediate Operand to Accumulator:

Encoding: 0011010w data
34 + w data

If w = 0, LSRC = AL, RSRC = data, DEST = AL.
If w = 1, LSRC = AX, RSRC = data, DEST = AX

Example: **Immediate (byte):**

```
XOR    AL,11110110B
XOR    AL,0F6H
```

Immediate (word):

```
XOR    AX,23F6H
XOR    AX,75Q
XOR    AX,23F6H   ;AX DESTINATION
```

XOR
Exclusive OR

Immediate Operand to Memory or Register Operand:

Encoding: 1000000w mod110r/m data
80 + w mod110r/m data

LSRC = EA, RSRC = data, DEST = EA

Example: **Immediate with register:**

```
XOR   AH,0F6H
XOR   CL,37
XOR   DI,23F5H
```

Immediate with memory:

```
XOR   MEM_BYTE,3DH
XOR   GAMMA[BX][DI],0FACEH
XOR   ALPHA[DI],VAL_EQU_33H
```

Another example:

```
BITMASK EQU     20H
FLAGS   DB      ?
        o
        o
        o
        XOR     FLAGS,BITMASK ;TOGGLE STATE
                              ; OF FLAG BIT
```

Index

Special Characters

.LALL 1-4, 1-10, 1-16, 2-6
.LALL pseudo-op 2-52
.LFCOND 1-4
.SALL pseudo-op 2-52
.SFCOND 1-4
.TFCOND 1-4
.XALL 1-16, 2-6
.XALL pseudo-op 2-52
.XCREF 2-12
.286C 1-17
.8086 1-17
.8087 1-17
& Special Macro Operator 1-14, 2-5
.CREF .XCREF 2-12
.LALL .SALL .XALL 2-52
.LFCOND (List False Conditionals) 2-53
.LIST .XLIST 2-54
.RADIX 2-69
.SFCOND 2-80
.TFCOND 2-84
.286C (Set 80286 Mode) 2-2
.8086 (Reset 80286 Mode) 2-3
.8087 (Set 8087-80287 Mode) 2-4
! Special Macro Operator 2-7
;; Special Macro Operator 2-6
/A, parameter 2-36
/D, parameter 2-54
/E parameter 1-17
/R parameter 1-17, 2-4
/S, parameter 2-36
/X, parameter 1-5, 1-6, 2-52, 2-84
% Special Macro Operator 1-15, 2-8
%OUT 1-4, 2-62
= Equal Sign 2-31
= Special Macro Operator 1-15

A

AAA ASCII Adjust for Addition 3-2
AAD ASCII Adjust for Division 3-3
AAM ASCII Adjust for Multiply 3-4
AAS ASCII Adjust for Subtraction 3-5
ADC Add with Carry 3-6
ADD Addition 3-8
Addressing Mode Byte 1-21
align-type
  byte 2-76
  page 2-77
  paragraph 2-76
  word 2-77
AND Logical AND 3-11
ASSUME 2-9
ASSUME nothing pseudo-op 2-9
ASSUME pseudo-op 2-9
at combine-type 2-77
attribute

  FAR 2-65
  NEAR 2-65

B

BOUND (80286) Check Array Index Against Bounds 3-14
byte align-type 2-76

C

CALL Call a Procedure 3-16
CBW Convert Byte to Word 3-20
class 2-78
CLC Clear Carry Flag 3-21
CLD Clear Direction Flag 3-22
CLI Clear Interrupt Flag (Disable) 3-23
CMC Complement Carry Flag 3-24
CMP Compare Two Operands 3-25
CMPS/CMPSB/CMPSW Compare Byte or Word String 3-28
combine-type
  at 2-77
  common 2-77
  public 2-77
COMMENT 1-10, 2-11
common combine-type 2-77
condition codes, 8087 3-142
Conditional pseudo-op 1-3
Conditionals, False 1-4
convention rules iii
CREF pseudo-op 2-12
CWD Convert Word to Doubleword 3-31

D

DAA Decimal Adjust for Addition 3-32
DAS Decimal Adjust for Subtraction 3-33
Data pseudo-op 1-4
DB Define Byte 2-13
DD Define Doubleword 2-15
DEC Decrement Destination by One 3-34
direction flag 3-28, 3-194, 3-263
directives 1-1, 2-1
DIV Division, Unsigned 3-36
DQ Define Quadword 2-17
DT Define Tenbytes 2-19
dummy 1-13
dummy list, MACRO pseudo-op 1-8
DW Define Word 2-21

E

ELSE 2-23
END 2-24
ENDIF 2-26
ENDM 1-9, 2-27
ENDP 2-28, 2-65
ENDS 2-29, 2-81

ENTER (80286) Make Stack Frame for Procedure Parameters 3-39
EQU 2-30
equal sign (=) pseudo-op 2-31
ESC Escape 3-41
EVEN 2-32
EXITM 2-33
EXTRN 2-34

F

FABS (8087) Absolute value 3-43
FADD (8087) Add Real 3-44
FADDP (8087) Add Real and Pop 3-47
False conditionals 1-4
FAR type attribute 2-65
FBLD (8087) Packed Decimal (BCD) Load 3-48
FBSTP (8087) Packed Decimal (BCD) Store and Pop 3-49
FCHS (8087) Change Sign 3-50
FCLEX (8087) Clear Exceptions 3-51
FCOM (8087) Compare Real 3-52
FCOMP (8087) Compare Real and Pop 3-55
FCOMPP (8087) Compare Real and Pop Twice 3-58
FDECSTP (8087) Decrement 8087 Stack Pointer 3-59
FDISI (8087) Disable Interrupts 3-60
FDIV (8087) Divide Real 3-61
FDIVP (8087) Divide Real and Pop 3-64
FDIVR (8087) Divide Real Reversed 3-65
FDIVRP (8087) Divide Real Reversed and Pop 3-68
FENI (8087) Enable Interrupts 3-69
FFREE (8087) Free Register 3-70
FIADD (8087) Integer Add 3-71
FICOM (8087) Integer Compare 3-72
FICOMP (8087) Integer Compare and Pop 3-74
FIDIV (8087) Integer Divide 3-76
FIDIVR (8087) Integer Divide Reversed 3-77
field, mode 1-21
field, r/m (register/memory field) 1-21
field, register 1-22
fields, instruction 1-21
FILD (8087) Integer Load 3-78
FIMUL (8087) Integer Multiply 3-80
FINCSTP (8087) Increment 8087 Stack Pointer 3-81
FINIT (8087) Initialize Processor 3-82
FIST (8087) Integer Store 3-83
FISTP (8087) Integer Store and Pop 3-84
FISUB (8087) Integer Subtract 3-86
FISUBR (8087) Integer Subtract Reversed 3-87
Flag Register 1-23
flag, clear direction 3-22

flag, set direction 3-272
Flags 1-23
FLD (8087) Load Real 3-88
FLDCW (8087) Load Control Word 3-91
FLDENV (8087) Load Environment 3-92
FLDLG2 (8087) Load $Log_{10}2$ 3-95
FLDLN2 (8087) Load Log base e of 2 3-96
FLDL2E (8087) Load Log_2e 3-93
FLDL2T (8087) Load $Log_2 10$ 3-94
FLDPI (8087) Load PI 3-97
FLDZ (8087) Load Zero 3-98
FLD1 (8087) Load +1.0 3-90
FMUL (8087) Multiply Real 3-99
FMULP (8087) Multiply Real and Pop 3-102
FNCLEX (8087) Clear Exceptions 3-103
FNDISI (8087) Disable Interrupts 3-104
FNENI (8087) Enable Interrupts 3-105
FNINIT (8087) Initialize Processor 3-106
FNOP (8087) No Operation 3-107
FNSAVE (8087) Save State 3-108
FNSTCW (8087) Store Control Word 3-109
FNSTENV (8087) Store Environment 3-110
FNSTSW (8087) Store Status Word 3-112
FPATAN (8087) Partial Arc Tangent 3-114
FPREM (8087) Partial Remainder 3-115
FPTAN (8087) Partial Tangent 3-117
FRNDINT (8087) Round to Integer 3-118
FRSTOR (8087) Restore State 3-119
FSAVE (8087) Save State 3-120
FSCALE (8087) Scale 3-121
FSQRT (8087) Square Root 3-122
FST (8087) Store Real 3-123
FSTCW (8087) Store Control Word 3-125
FSTENV (8087) Store Environment 3-126
FSTP (8087) Store Real and POP 3-127
FSTSW (8087) Store Status Word 3-129
FSUB (8087) Subtract Real 3-130
FSUBP (8087) Subtract Real and POP 3-133
FSUBR (8087) Subtract Real Reversed 3-134
FSUBRP (8087) Subtract Real Reversed and POP 3-138
FTST (8087) Test 3-139
FWAIT (8087) Wait (CPU Instruction) 3-140
FXAM (8087) Examine 3-142
FXCH (8087) Exchange Registers 3-144
FXTRACT (8087) Extract Exponent and Significand 3-146
FYL2X (8087) Y * Log_2X 3-147
FYL2XP1 (8087) Y * $Log_2(X+1)$ 3-148

F2XM1 (8087) 2 to the X power -1 3-42

G

GROUP 2-36

H

HLT Halt 3-149

I

IDIV Integer Division, Signed 3-150
IFxxxx Conditional Pseudo-ops 2-40
IMUL (80286) Integer Immediate Multiply 3-155
IMUL Integer Multiply 3-153
IN Input Byte or Word 3-157
INC Increment Destination by One 3-159
INCLUDE 1-11, 2-45
INS/INSB/INSW (80286) Input from Port to String 3-161
Instruction
    addressing mode byte 1-21
    description 1-23
    fields 1-21
    format 1-21
    mode field 1-21
    notations 1-23
    operation code byte 1-21
    register field 1-22
    register/memory field 1-22
    symbols 1-23
Instructions 1-19
    AAA/ASCII Adjust for Addition 3-2
    AAD/ASCII Adjust for Division 3-3
    AAM/ASCII Adjust for Multiply 3-4
    AAS/ASCII Adjust for Subtraction 3-5
    ADC/Add with Carry 3-6
    ADD/Addition 3-8
    Addressing Mode Byte 1-21
    AND/Logical AND 3-11
    BOUND (80286)/Check Array Index Against Bounds 3-14
    CALL/Call a Procedure 3-16
    CBW/Convert Byte to Word 3-20
    CLC/Clear Carry Flag 3-21
    CLD/Clear Direction Flag 3-22
    CLI/Clear Interrupt Flag (Disable) 3-23
    CMC/Complement Carry Flag 3-24
    CMP/Compare Two Operands 3-25
    CMPS/CMPSB/CMPSW Compare Byte or Word String 3-28
    CWD/Convert Word to Doubleword 3-31

DAA/Decimal Adjust for Addition 3-32
DAS/Decimal Adjust for Subtraction 3-33
DEC/Decrement Destination by One 3-34
DIV/Division, Unsigned 3-36
ENTER (80286)/Make Stack Frame for Procedure Parameters 3-39
ESC/Escape 3-41
FABS (8087)/Absolute value 3-43
FADD (8087)/Add real 3-44
FADDP (8087)/Add real and pop 3-47
FBLD (8087)/Packed decimal (BCD) load 3-48
FBSTP Packed decimal (BCD) store and pop 3-49
FCHS (8087)/Change sign 3-50
FCLEX (8087)/Clear exceptions 3-51
FCOM (8087)/Compare real 3-52
FCOMP (8087)/Compare real and pop 3-55
FCOMPP (8087)/Compare real and pop twice 3-58
FDECSTP (8087)/Decrement 8087 stack pointer 3-59
FDISI (8087)/Disable interrupts 3-60
FDIV (8087)/Divide real 3-61
FDIVP (8087)/Divide real and pop 3-64
FDIVR (8087)/Divide real reversed 3-65
FDIVRP (8087)/Divide real reversed and pop 3-68
FENI (8087)/Enable interrupts 3-69
FFREE (8087)/Free register 3-70
FIADD (8087)/Integer add 3-71
FICOM (8087)/Integer compare 3-72
FICOMP (8087)/Integer compare and pop 3-74
FIDIV (8087)/Integer divide 3-76
FIDIVR (8087)/Integer divide reversed 3-77
fields 1-21
FILD (8087)/Integer load 3-78
FIMUL (8087)/Integer multiply 3-80
FINCSTP (8087)/Increment 8087 stack pointer 3-81
FINIT (8087)/Initialize processor 3-82
FIST (8087)/Integer store 3-83
FISTP (8087)/Integer store and pop 3-84
FISUB (8087)/Integer subtract 3-86
FISUBR (8087)/Integer subtract reversed 3-87
FLD (8087)/Load real 3-88
FLDCW (8087)/Load control word 3-91
FLDENV (8087)/Load environment 3-92

FLDLG2 (8087)/Load $Log_{10}2$ 3-95
FLDLN2 (8087)/Load Log base e of 2 3-96
FLDL2E (8087)/Load Log_2e 3-93
FLDL2T (8087)/Load Log_210 3-94
FLDPI (8087)/Load PI 3-97
FLDZ (8087)/Load Zero 3-98
FLD1 (8087)/Load +1.0 3-90
FMUL (8087)/Multiply Real 3-99
FMULP (8087)/Multiply Real and Pop 3-102
FNCLEX (8087)/Clear Exceptions 3-103
FNDISI (8087)/Disable Interrupts 3-104
FNENI (8087)/Enable Interrupts 3-105
FNINIT (8087)/Initialize Processor 3-106
FNOP (8087)/No Operation 3-107
FNSAVE (8087)/Save State 3-108
FNSTCW (8087)/Store Control Word 3-109
FNSTENV (8087)/Store Environment 3-110
FNSTSW (8087)/Store Status Word 3-112
format 1-21
FPATAN (8087)/Partial Arc Tangent 3-114
FPREM (8087)/Partial Remainder 3-115
FPTAN (8087)/Partial Tangent 3-117
FRNDINT (8087)/Round to Integer 3-118
FRSTOR (8087)/Restore State 3-119
FSAVE (8087)/Save State 3-120
FSCALE (8087)/Scale 3-121
FSQRT (8087)/Square Root 3-122
FST (8087)/Store Real 3-123
FSTCW (8087)/Store Control Word 3-125
FSTENV (8087)/Store Environment 3-126
FSTP (8087)/Store Real and POP 3-127
FSTSW (8087)/Store Status Word 3-129
FSUB (8087)/Subtract Real 3-130
FSUBP (8087)/Subtract Real and POP 3-133
FSUBR (8087)/Subtract Real Reversed 3-134
FSUBRP (8087)/Subtract Real Reversed and POP 3-138
FTST (8087)/Test 3-139
FWAIT (8087)/Wait (CPU Instruction) 3-140
FXAM (8087)/Examine 3-142
FXCH (8087)/Exchange Registers 3-144
FXTRACT (8087)/Extract Exponent and Significand 3-146
FYL2X (8087)/Y * Log_2X 3-147
FYL2XP1 (8087)/Y * $Log_2(X+1)$ 3-148

F2XM1 (8087)/2 to the X power -1 3-42
HLT/Halt 3-149
IDIV/Integer Division, Signed 3-150
IMUL (80286)/Integer Immediate Multiply 3-155
IMUL/Integer Multiply 3-153
IN/Input Byte or Word 3-157
INC/Increment Destination by One 3-159
INS/INSB/INSW (80286)/Input from Port to String 3-161
INT/Interrupt 3-163
INTO/Interrupt If Overflow 3-165
IRET/Interrupt Return 3-167
J(condition)/Jump Short If Condition Met 3-168
JA/Jump if Above 3-169
JAE/Jump If Above or Equal 3-169
JB/Jump If Below 3-169
JBE/Jump If Below Or Equal 3-169
JC/Jump If Carry 3-169
JCXZ/Jump If CX Is Zero 3-169
JE/Jump If Equal 3-169
JG/Jump If Greater 3-169
JGE/Jump If Greater or Equal 3-169
JL/Jump If Less 3-169
JLE/Jump If Less or Equal 3-169
JMP/Jump 3-171
JNA/Jump If Not Above 3-169
JNAE/Jump If not Above nor Equal 3-169
JNB/Jump If Not Below 3-169
JNBE/Jump if Not Below or Equal 3-169
JNC/Jump If No Carry 3-169
JNE/Jump If Not Equal 3-169
JNG/Jump If Not Greater 3-169
JNGE/Jump If Not Greater nor Equal 3-169
JNL/Jump If Not Less 3-169
JNLE/Jump If Not Less nor Equal 3-169
JNO/Jump If No overflow 3-169
JNP/Jump If No Parity 3-169
JNS/Jump If No Sign/If Positive 3-169
JNZ/Jump If Not Zero 3-169
JO/Jump On Overflow 3-169
JP/Jump On Parity 3-169
JPE/Jump If Parity Even 3-169
JPO/Jump If Parity Odd 3-169
JS/Jump On Sign 3-169
JZ/Jump If Zero 3-169
LAHF/Load AH from Flags 3-174
LDS/Load Data Segment Register 3-175
LEA/Load Effective Address 3-176

LEAVE (80286)/High Level Procedure Exit 3-177
LES/Load Extra Segment Register 3-178
LOCK/Lock Bus 3-179
LODS/LODSB/LODSW/ Load Byte or Word String 3-180
LOOP/Loop Until Count Complete 3-183
LOOPE/LOOPZ/Loop If Equal/If Zero 3-185
LOOPNE/LOOPNZ/Loop If Not Equal/Not Zero 3-187
mode field 1-21
MOV/Move 3-189
MOVS/MOVSB/MOVSW/ Move Byte or Word String 3-194
MUL/Multiply, Unsigned 3-197
NEG/Negate, Form Twos Complement 3-199
NOP/No Operation 3-201
NOT/Logical Not 3-202
OR/Logical Inclusive Or 3-204
OUT/Output Byte or Word 3-207
OUTS/OUTSB/OUTSW (80286)/Output String to Port 3-209
POP/Pop Word Off Stack to Destination 3-211
POPA (80286)/Pop All General Registers 3-213
POPF/Pop Flags Off Stack 3-214
PUSH (80286)/Push Immediate onto Stack 3-217
PUSH/Push Word onto Stack 3-215
PUSHA (80286)/Push All General Registers 3-218
PUSHF/Push Flags onto Stack 3-219
RCL (80286)/Rotate Left Through Carry 3-223
RCL/Rotate Left Through Carry 3-220
RCR (80286)/Rotate Right Through Carry 3-229
RCR/Rotate Right Through Carry 3-226
register field 1-22
register/memory field 1-22
REP/Repeat String Operation 3-232
REPE/Repeat String Operation 3-232
REPNE/Repeat String Operation 3-232
REPNZ/Repeat String Operation 3-232
REPZ/Repeat String Operation 3-232
RET/Return from Procedure 3-235
ROL (80286)/Rotate Left 3-240
ROL/Rotate Left 3-238
ROR (80286)/Rotate Right 3-245
ROR/Rotate Right 3-243
SAHF/Store AH in Flags 3-248
SAL (80286)/Shift Arithmetic Left 3-252
SAL/Shift Arithmetic Left 3-249
SAR (80286)/Shift Arithmetic Right 3-257

  SAR/Shift Arithmetic Right 3-255
  SBB/Subtract with Borrow 3-260
  SCAS/SCASB/SCASW/ Scan Byte or Word String 3-263
  SHL (80286)/Shift Logical Left 3-252
  SHL/Shift Logical Left 3-249
  SHR (80286)/Shift Logical Right 3-268
  SHR/Shift Logical Right 3-266
  STC/Set Carry Flag 3-271
  STD/Set Direction Flag 3-272
  STI/SET Interrupt Flag (Enable) 3-273
  STOS/STOSB/STOSW/ Store Byte or Word String 3-274
  SUB/Subtract 3-277
  TEST/Test (Logical Compare) 3-280
  WAIT/Wait 3-283
  XCHG/Exchange 3-284
  XLAT/Translate 3-286
  XOR/Exclusive OR 3-287
Instructions, 80286 1-20
Instructions, 80287 1-20
Instructions, 8087 1-19
INT Interrupt 3-163
INTO Interrupt If Overflow 3-165
IRET Interrupt Return 3-167
IRP 2-47
IRPC 2-49

J

J(condition) Jump Short If Condition Met 3-168
JA/Jump if Above 3-169
JAE/Jump If Above or Equal 3-169
JB/Jump If Below 3-169
JBE/Jump If Below Or Equal 3-169
JC/Jump If Carry 3-169
JCXZ/Jump If CX Is Zero 3-169
JE/Jump If Equal 3-169
JG/Jump If Greater 3-169
JGE/Jump If Greater or Equal 3-169
JL/Jump If Less 3-169
JLE/Jump If Less or Equal 3-169
JMP Jump 3-171
JNA/Jump If Not Above 3-169
JNAE/Jump If not Above nor Equal 3-169
JNB/Jump If Not Below 3-169
JNBE/Jump if Not Below or Equal 3-169
JNC/Jump If No Carry 3-169
JNE/Jump If Not Equal 3-169
JNG/Jump If Not Greater 3-169
JNGE/Jump If Not Greater nor Equal 3-169
JNL/Jump If Not Less 3-169
JNLE/Jump If Not Less nor Equal 3-169
JNO/Jump If No overflow 3-169
JNP/Jump If No Parity 3-169

JNS/Jump If No Sign/If Positive 3-169
JNZ/Jump If Not Zero 3-169
JO/Jump On Overflow 3-169
JP/Jump On Parity 3-169
JPE/Jump If Parity Even 3-169
JPO/Jump If Parity Odd 3-169
JS/Jump On Sign 3-169
JZ/Jump If Zero 3-169

L

LABEL 2-50
LAHF Load AH from Flags 3-174
LALL pseudo-op 2-52
LDS Load Data Segment Register 3-175
LEA Load Effective Address 3-176
LEAVE (80286) High Level Procedure Exit 3-177
LES Load Extra Segment Register 3-178
LFCOND pseudo-op 1-5, 2-53
LIST pseudo-op 2-54
Listing pseudo-ops 1-4
listing, page size 2-63
LOCAL 1-10, 2-55
LOCK Lock Bus 3-179
LODS/LODSB/LODSW Load Byte or Word String 3-180
LOOP Loop Until Count Complete 3-183
LOOPE/LOOPZ Loop If Equal/If Zero 3-185
LOOPNE/LOOPNZ Loop If Not Equal/If Not Zero 3-187

M

MACRO 2-56
Macro pseudo-ops 1-8
MASK operator 2-71, 2-74
memory combine-type 2-78
    memory 2-78
MODE command, printer 2-64
mode field 1-21
Mode pseudo-ops 1-17
MOV Move 3-189
MOVS/MOVSB/MOVSW Move Byte or Word String 3-194
MUL Multiply, Unsigned 3-197

N

NAME 2-60
NEAR type attribute 2-65
NEG Negate, Form Twos Complement 3-199
NOP No Operation 3-201
NOT Logical Not 3-202

O

operand list, MACRO pseudo-op 1-8
operations, pseudo-op 1-1, 2-1
operator
  MASK 2-71, 2-74
  width 2-74
OR Logical Inclusive Or 3-204
ORG 2-61
OUT (%OUT) 2-62
OUT Output Byte or Word 3-207
OUTS/OUTSB/OUTSW (80286) Output String to Port 3-209

P

PAGE 2-63
page align-type 2-76
para align-type 2-76
parameter /A 2-36
parameter /D 2-54
parameter /S 2-36
parameter /X 1-5, 1-6, 2-52, 2-84
parameter list, MACRO pseudo-op 1-8
parameter, /E 1-17
parameter, /R 1-17, 2-4
parameters, MACRO pseudo-op 1-8
parameters, pseudo-op 1-9
POP Pop Word Off Stack to Destination 3-211
POPA (80286) Pop All General Registers 3-213
POPF Pop Flags Off Stack 3-214
PROC 2-65
pseudo-ops
  .CREF 2-12
  .LALL 2-52
  .LFCOND 1-5
  .LFCOND (List False Conditionals) 2-53
  .LIST 2-54
  .RADIX 2-69
  .SALL 2-52
  .SFCOND 1-5, 2-80
  .TFCOND 1-5, 2-84
  .XALL 2-52
  .XCREF 2-12
  .XLIST 2-54
  .286C (Set 80286 Mode) 2-2
  .8086 (Reset 80286 Mode) 2-3
  .8087 (Set 8087-80287 Mode) 2-4
  & macro operator 2-5
  ! macro operator 2-7
  ;; macro operator 2-6
  % macro operator 2-8
  %OUT 2-62
  = (equal sign) 2-31
  ASSUME 2-9
  ASSUME NOTHING 2-9
  COMMENT 2-11
  Conditional 1-3
  Data 1-4
  DB (define byte) 2-13
  DD (define Doubleword) 2-15
  DQ (define Quadword) 2-17
  DT (define Tenbytes) 2-19
  DW (define word) 2-21
  ELSE 2-23
  END 2-24

  - ENDIF 2-26
  - ENDM 2-27
  - ENDP 2-28
  - ENDS 2-29
  - EQU 2-30
  - EVEN 2-32
  - EXITM 2-33
  - EXTRN 2-34
  - GROUP 2-36
  - IFxxxx 2-40
  - INCLUDE 2-45
  - IRP 2-47
  - IRPC 2-49
  - LABEL 2-50
  - Listing 1-4
  - LOCAL 2-55
  - MACRO 1-8, 2-56
  - Mode 1-17
  - NAME 2-60
  - ORG 2-61
  - OUT (%OUT) 2-62
  - PAGE 2-63
  - PROC 2-65
  - PUBLIC 2-67
  - PURGE 2-68
  - RECORD 2-71
  - REPT 2-75
  - SEGMENT 2-76
  - STRUC 2-81
  - SUBTTL 2-83
  - TITLE 2-85
- PUBLIC 2-67
- public combine-type 2-77
- PURGE 2-68
- PUSH (80286) Push Immediate onto Stack 3-217
- PUSH Push Word onto Stack 3-215
- PUSHA (80286) Push All General Registers 3-218
- PUSHF Push Flags onto Stack 3-219

R

- r/m field (register/memory field) 1-21
- RADIX pseudo-op 2-69
- RCL (80286) Rotate Left Through Carry 3-223
- RCL Rotate Left Through Carry 3-220
- RCR (80286) Rotate Right Through Carry 3-229
- RCR Rotate Right Through Carry 3-226
- RECORD 2-71
- register field 1-22
- register, Flag 1-23
- REP/REPZ/REPE/REPNE/REPNZ Repeat String Operation 3-232
- Repeat Block pseudo-ops 1-8
- REPT 2-75
- RET Return from Procedure 3-235
- ROL (80286) Rotate Left 3-240
- ROL Rotate Left 3-238
- ROR (80286) Rotate Right 3-245
- ROR Rotate Right 3-243
- rules, convention iii

S

- SAHF Store AH in Flags 3-248

INDEX

SAL/SHL (80286) Shift Arithmetic Left/Shift Logical Left 3-252
SAL/SHL Shift Arithmetic Left/Logical Left 3-249
SALL pseudo-op 2-52
SAR (80286) Shift Arithmetic Right 3-257
SAR Shift Arithmetic Right 3-255
SBB Subtract with Borrow 3-260
SCAS/SCASB/SCASW Scan Byte or Word String 3-263
SEGMENT 2-76
    pseudo-op 2-76
set direction flag 3-272
SFCOND pseudo-op 1-5, 2-80
SHR (80286) Shift Logical Right 3-268
SHR Shift Logical Right 3-266
special
    macro forms 2-5, 2-6, 2-7, 2-8
    macro operators 2-5, 2-6, 2-7, 2-8
stack combine-type 2-78
    stack 2-78
STC Set Carry Flag 3-271
STD Set Direction Flag 3-272
STI Set Interrupt Flag (Enable) 3-273
STOS/STOSB/STOSW Store Byte or Word String 3-274
STRUC 2-81
SUB Subtract 3-277
SUBTTL 2-83

T

TEST Test (Logical Compare) 3-280
TFCOND pseudo-op 1-5, 2-84
TITLE 2-85

W

WAIT Wait 3-283
width operator, usage 2-74
word align-type 2-76

X

XALL pseudo-op 2-52
XCHG Exchange 3-284
XCREF pseudo-op 2-12
XLAT Translate 3-286
XLIST pseudo-op 2-54
XOR Exclusive OR 3-287

Numerals

286C pseudo-op 2-2
80286 Instructions 1-20
80286-based IBM Personal Computers 1-17
80287 Instructions 1-20

80287 Math Co-Processor feature 1-17
8086 pseudo-op 2-3
8087 Instructions 1-19
8087 Math Co-Processor feature 1-17
8087 pseudo-op 2-4
8088-based IBM Personal Computers 1-17

IBM

The Personal Computer
Computer Language
Series

Reader's Comment Form

Macro Assembler 6361620
Reference

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

The Personal Computer
Computer Language
Series

Reader's Comment Form

Macro Assembler 6361620
Reference

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

© IBM Corp. 1984
All rights reserved.

International Business
Machines Corporation
P.O. Box 1328-S
Boca Raton, Florida 33432

Printed in the
United States of America

6361620